A SEMANTIC MODEL FOR ORTHOHEDRAL PRISMATIC PART DESIGN AND MANUFACTURE

By

Srinivasan Sridhar

B.S. Mechanical Engineering
The University of Mysore, 1984

A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

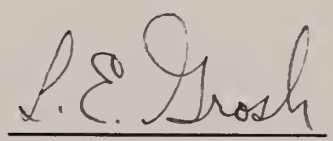
MASTER OF SCIENCE

Department of Industrial Engineering

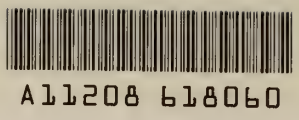
KANSAS STATE UNIVERSITY
Manhattan, Kansas

1989

Approved By:



Major Professor



CONTENTS

ACKNOWLEDGEMENTS

I would like to thank Dr.L. E. Grosh, Professor, Department of Industrial Engineering, for his support during this study.

I would like to thank Dr. Bradley Kramer, Assistant Professor, Department of Industrial Engineering, for his valuable directions during the entire course of this study.

My thanks also go out to Dr. Prakash Krishnaswami, Assistant Professor, Department of Mechanical Engineering, for his helpful design suggestions.

LIST OF FIGURES

LIST IF FIGURES (continued)

1. INTRODUCTION

The focus of this thesis is to describe the development of a prototype system for the design of a set of prismatic components with features. The problems with part representation using computers are also addressed. A semantic model of the system to design and manufacture components is developed and presented. The semantic model addresses key issues in the process of design and manufacture and explains the methodology adopted by the system.

Computer Aided Design (CAD) attempts to represent the geometric data of a component and manufacturing systems are the major users of this data. At the present time, a representation scheme that permits easy inference of designs by manufacturing systems has not been realized. This is primarily due to the nature of data that is represented by CAD systems. CAD systems view design representations in terms of the final geometries of components. The manufacturing departments view the end representation in terms of volumes removed or operations performed in the process of attaining the final design. The opposite views taken by design and manufacturing systems has made communications a difficult task. The key to attain the goal of automated CAD/CAM systems is a clear cut communications protocol that explicitly defines practices to be followed.

A standard means to communicate design and product data has not been realized yet; the underlying problem is in the representation of components. The development of stand-alone CAD, CAM and CAPP systems has been the major hurdle to establishing standards of communication between design and manufacturing systems. The beginning of communications between CAD and CAM systems is at the information root level. Practices and data operations followed by the design system should be compatible with the manufacturing system. The ideal situation is a design system producing data solely for the manufacturing system. The most important step towards process integration is a common data format followed by component systems.

Computer aided design and manufacture is especially undergoing vast changes due to the large potential to improve standards and practices.

1.1 A BRIEF HISTORY OF DESIGN AND MANUFACTURING PRACTICES

Traditionally, the creation of mechanical components involves several distinct phases which are functions of various departments. Conceptualization of a product is the first step in the creation of a component. Once a product is selected for manufacture, it goes through several phases before a blueprint or design is achieved. After a

product's conception, its functional requirements and mechanical requirements are determined based on the application area of the component. The manufacturing methods are then determined based on the design and material requirements.

1.1.1 Representation of Design

The component selected for manufacture has to be represented by some means in order to communicate the product design to the various organizational functions which will evaluate, analyze, manufacture, market and distribute the part. The medium of representation used before the development of computer tools was mechanical drawings which represented the design of the component to scale. Other details of the product such as material, tolerance and surface finish were represented notationally as additional requirements to the geometric characteristics of the component.

The manufacturing department was the primary user of the design drawing. Manufacturing engineers would interpret the mechanical drawing to understand the product design in order to find efficient means of manufacture of the component to design specifications. The engineering department also used the design drawing to ensure that the physical specifications of a component met the functional requirements.

1.1.2 Determining the design's functional reliability

Engineering analysis systems, such as finite element analysis and thermal analysis are used to ensure that a product's physical requirements meet the functional demands of the product. The output from the design phase is input to analysis systems. Engineering changes are suggested to the design as a result of analysis. The flow of data from the design to the engineering phase was traditionally by means of blueprints which were analyzed by engineering algorithms. The traditional output of engineering analysis was blueprints or mechanical drawings with suggested design changes.

1.1.3 Process Planning for the designed part

The process plan determined the operations selected to manufacture the designed component. It is one of the most important tasks in the manufacturing cycle. The process plan was traditionally tailored to the designed product for repetitive manufacture. Once a process or set of manufacturing operations was finalized, it was very difficult to effect any changes in the process plan.

A process plan would include the top-down description for each of the operations for the manufacture. The specifications from a process plan would include the design of the

component, the raw material, tolerances, operation sequence, and machine loading sequence. This information would be the input to the manufacture of the part.

1.1.4 Manufacture

Manufacture is the actual conversion of raw material to the finished component. The original design and the process plan are inputs to the manufacturing department. Manufacturing is the physical creation of the component meeting all specifications of designed component by following the process plan. A production plan following this activity would detail the sequence of parts for manufacture, parts batch size, and other industrial engineering functions.

1.2 THE IMPACT OF COMPUTERS IN DESIGN AND MANUFACTURING

The introduction of computers has brought about dramatic changes in product design and manufacturing. The following are the changes to the activities in Section 1.1 which have followed the introduction of computers (Figure 1.1).

1.2.1 Computer Aided Design and Drafting (CAD)

This task uses computers to generate product designs at a graphical workstation. Many present day CAD systems are capable of and used for drafting purposes only. CAD has

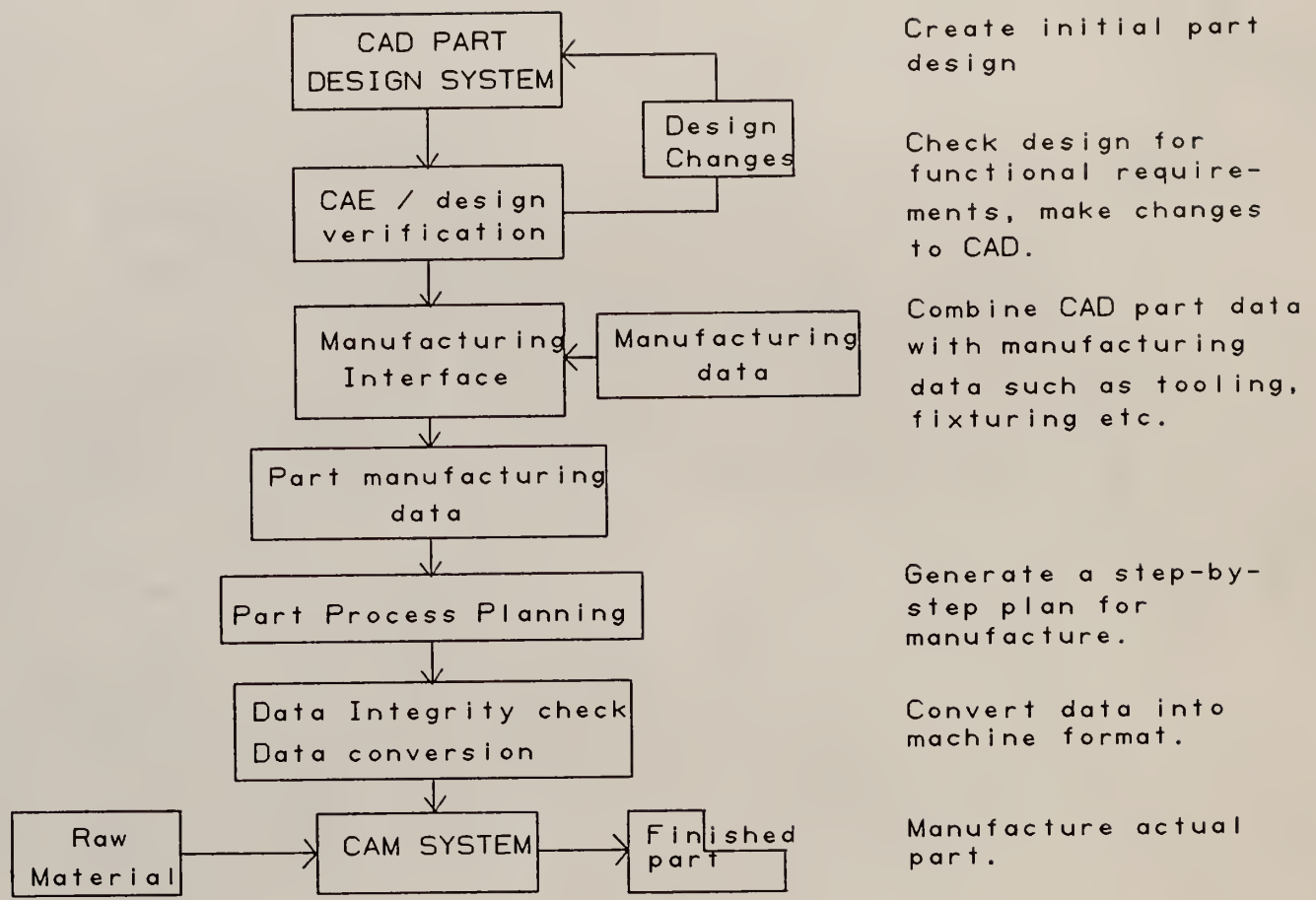


FIGURE 1.1 COMPUTERS IN DESIGN AND MANUFACTURING

given designers and draftsmen efficient tools for the automation of creating mechanical drawings. CAD systems were originally capable of creating only two dimensional geometry. More recently, CAD systems have been developed which are capable of three dimensional modeling. State-of-the-art CAD systems offer a host of other features to represent a component's geometric attributes.

1.2.2 Computer Aided Engineering (CAE)

Corresponding to the task of engineering analysis is CAE. CAE systems evaluate designs for their structural soundness and various other engineering properties. The overall strength characteristics are determined by CAE systems using computerized tools. Typical of CAE systems are finite element mesh analysis systems and thermal analysis systems which are used to determine the functional properties of a component.

1.2.3 Computer Aided Process Planning (CAPP)

The task of determining precise plans to manufacture the designed component is performed by computer aided process planning. Included in a process plan are the sequence of operations for manufacture, tool and equipment selection, and jigs and fixtures selection for workholding. It is the task of a computer aided process planner to exhaustively specify how the component is to be manufactured.

Computer aided process planning is a multidisciplinary activity involving the tasks of operation selection, machine selection, and conversion of the design to a machine tool understandable format. General purpose process planning is a highly dynamic activity. It has to operate on varying designs, select operations suitable for design, select from a multitude of machine tools, etc. CAPP systems will ideally output the initial manufacturing specifications in a data format that can be decoded by computers operating a CAM system. The present state of automatic process planning for generic components is underdeveloped.

1.2.4 Computer Aided Manufacture

A computer performs the task of data input to a machine tool in a CAM system. CAM begins with conversion of raw data to a format understandable by a machine tool for manufacturing. Numerically controlled (NC) machines are used in CAM. In CAM, a single machine or a host of machines may be controlled by computers. The computers controlling the machine tools to manufacture products perform a host of activities such as data input to the machine tool, control of data input to specific machines and the setting up of machines for the manufacture of specific components. Thus CAM systems offer flexibility with minimum setup time to manufacture a large variety of components.

1.3 THE ISLANDS OF AUTOMATION

The design and manufacturing phases have traditionally been distinctly separated. To automate the design/manufacturing phases, it is necessary to find ways of integrating the four different activities outlined in 1.2. A full automation of all activities is an ambitious goal. There are several problems to be solved before full integration becomes a reality.

The first two tasks of CAE and CAD are in a reasonable stage of automation. CAE and CAD systems communicate to each other by mutually understandable data formats. However, the transition of data from CAD to CAPP and to CAM still remains elusive. This is because a general purpose CAPP system to convert CAD requirements to CAM functions has not yet been realized, though several projects within a narrow domain of applications and geometric part descriptions have been successful [1][2] (Figure 1.2).

1.4 THE IMPORTANCE OF PRODUCT DATA

The ability to communicate computer generated data effectively is a pressing need in today's CAD/CAM community. Many CAD/CAM systems of varying sizes and capabilities exist, but none are compatible with each other[3].

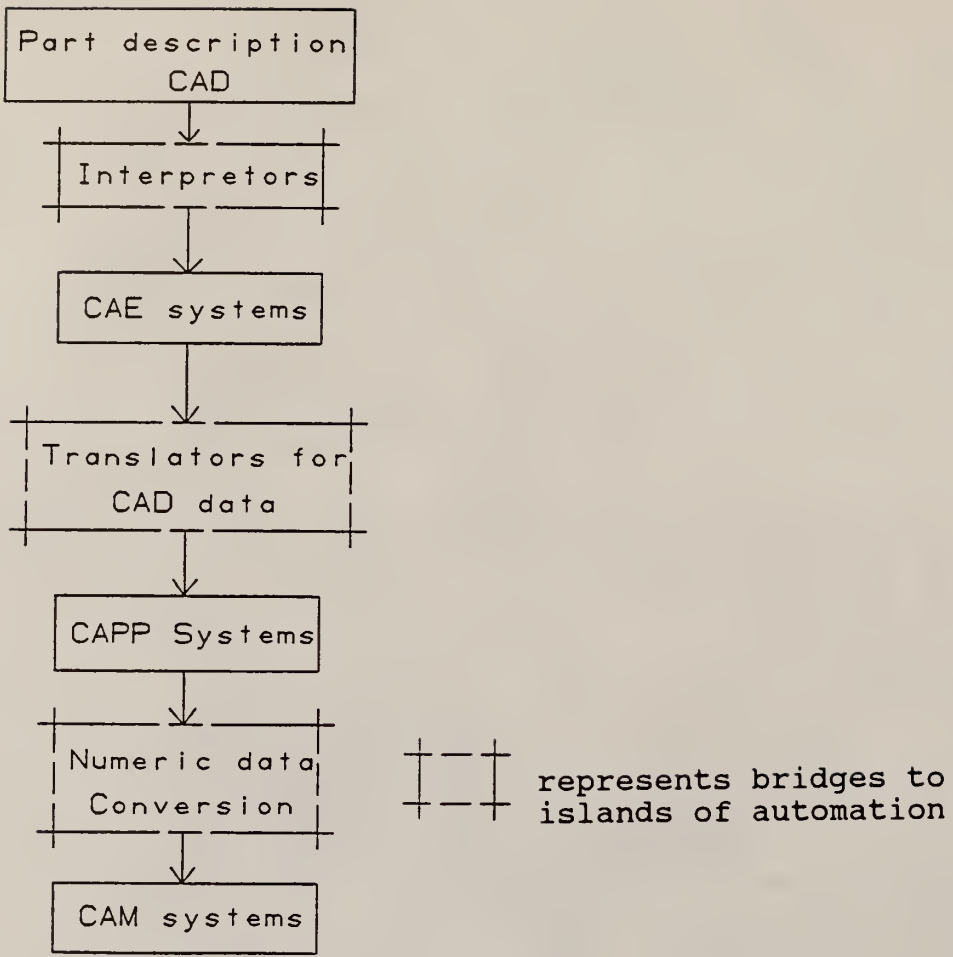


FIGURE 1.2 THE ISLANDS OF AUTOMATION

Before the advent of CAD/CAM systems, product design information was communicated from design to manufacturing by means of classical blueprints. However, not all the information can be captured on a single blueprint. The blueprint medium of communication has the drawback of manual transfer of data. The biggest drawback of blueprints is that human interpretation is required at every stage. Any engineering change involves the tedious process of interpreting a blueprint and extracting data. Similarly, data is interpreted manually for manufacturing functions. A common data representation in a form usable by all departments is a basic requirement for automating the process. This also serves to reduce the repetition of design interpretations.

Compared to blueprints, CAD part representation can be changed easily and are in a more easily readable format. They are also easier to transmit, file and recall. Functions in CAD to query a component, generate cross sections and magnify sections are highly useful tools unavailable with blueprint representation.

Once the design of a component is formalized, the data must be communicated to other departments who will use the data. A standard format of data capturing all the vital information regarding the product is called product data format.

To automate the functions involved in engineering design and manufacturing process planning, a general purpose parts description method suitable for all design and planning requirements is essential. The method must be capable of describing part geometry and form and must permit rapid retrieval and display. The product data must be structured so that association of parts to assemblies and to manufacturing processes can be made and changed dynamically[3]. Ideally, a standard format of product data used by various vendors of CAD and CAM systems will solve most product data problems. Attempts to standardize data have met with little success with most vendors who follow their own form of data representation.

1.5 DATA REPRESENTATION AND METHODS OF IMPLEMENTATION

One of the reasons that the islands of automation exist is that the part representation is focused in a narrow domain. Present day CAD systems are highly system specific. Stand alone systems output excellent results but when required to integrate with other systems they turn out to be inefficient. Efficient part representation methods/schemes will provide solutions to the islands problem. Present day part representation schemes are presented in this section and their applicability to process integration is described.

1.5.1 Boundary Representations (BREP)

BREP refers to boundary representation. Part geometry is represented in this scheme by means of faces, edges and vertices (commonly called FEV representation). In this system, every item is represented by its composite faces, edges and vertices and a numbering scheme is adopted for the three attributes. A BREP datafile consists of identifiers and geometric coordinate points for each of the vertices in a part, an array consisting of identifiers for edges with corresponding pairs of vertices and an array of attributes for faces.

A serious shortcoming in BREP for CAM applications is the shortage of information contained in data fields. Parameters such as surface finish, tolerances and material may exist only as character strings. A CAPP system requires such crucial data for planning operations to machine a part. There does not seem to be any standard form of representation of such data. BREP techniques suit CAD applications and offer excellent features for geometric manipulation and display functions.

1.5.2 Constructive Solid Geometry

Constructive solid geometry (CSG) is similar to BREP in it's approach to part definition. BREP defines parts by the boundary elements and CSG defines parts by the volume occupied. CSG systems of part description suffer from the same drawbacks as BREP for part modeling for CAM. They represent only geometric data and the lack of manufacturing data is the main deficiency for manufacturing purposes.

1.5.3 Feature Based Modeling (Figure 1.3)

Features are the next conceptually higher step towards product data definition. Feature based design provides details of the following: graphic representation, manufacturing details, material and tool selection and machine selection. Features are user defined types and the bounds to entity representation and description are endless. Thus a feature is a geometric form or entity:

1. Whose presence or dimensions are required to perform at least one CIM function (such as graphics, process planning, analysis and manufacturability evaluation), and
2. Whose availability as a primitive permits the design process to occur[2].

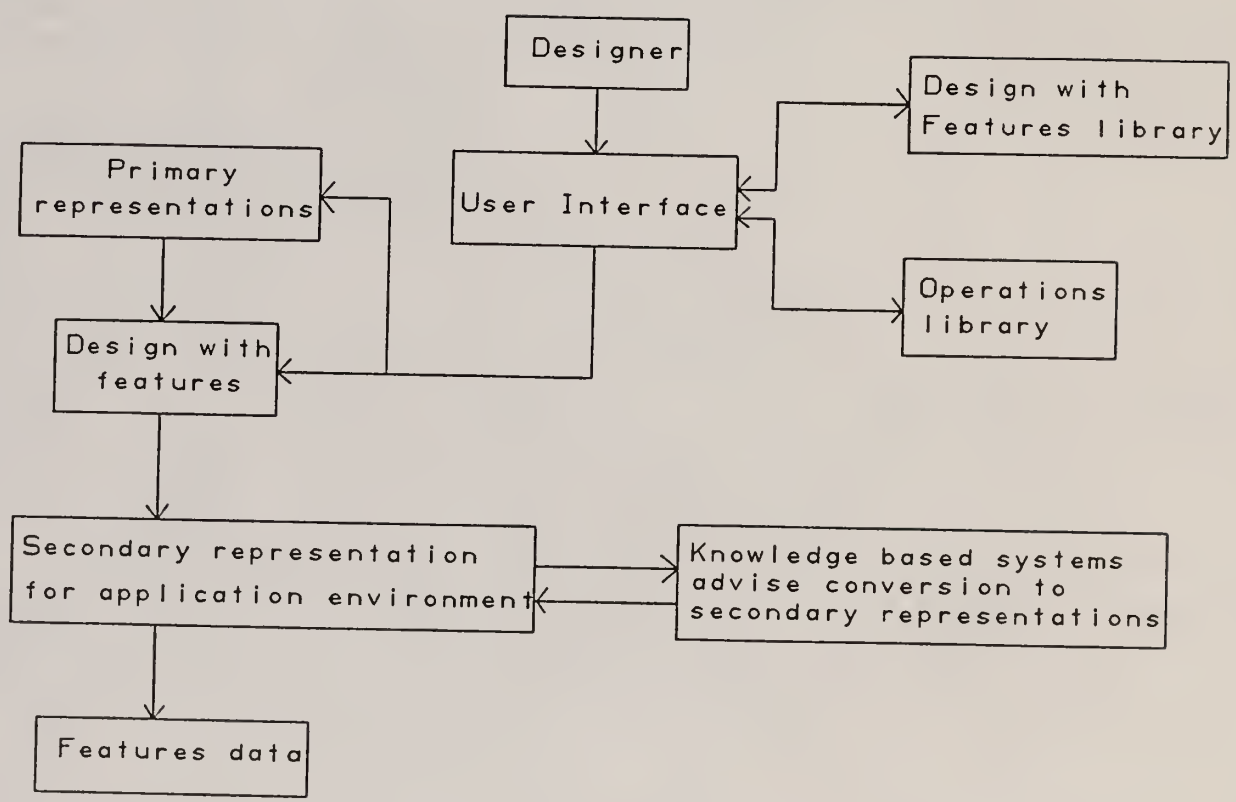


FIGURE 1.3 FEATURE BASED DESIGN SYSTEMS

In short, a feature is a named entity that has attributes of both form and function. Feature based systems would exist as computer processes with little or no human intervention. Fully developed feature based systems would link CAD part descriptions to CAM processes by performing automatic part interpretation and generate information necessary to manufacture the part. A set of algorithms describing rules to interpret CAD part description and convert to a CAM readable format forms the heart of the features system. Methods of revision of part geometry and process can be made easier by making the features system available to the part designer and the manufacturer. Several research studies are being conducted examining the capabilities of features systems for manufacturing environments.

1.6 A SUMMARY OF MODELS DISCUSSED

Although design database formats and methods described and standardized above have improved the ability for computers to store more complete design information, these databases do not necessarily reflect the data in a format readily usable by manufacturing systems. The reason for this is that the functions of design and manufacturing are different. Design systems produce data without regard for manufacturing functions. Output design data is in terms of final specifications. Manufacturing operations require data

in intermediate steps of material removal[10].

In using standard forms of data available, operations in manufacturing are not completely defined. Data in it's primitive form for manufacturing should contain information about form and function. In a hierarchical fashion, such data will carry rich semantics describing vital data for manufacturing.

Current efforts are underway to develop data models that communicate product data through neutral file formats with the PDES (Product Data Exchange Specifications). Human interpretation is still necessary at this stage of PDES development for data to be used by CAM systems. However, research in PDES is still being conducted and it is hoped that PDES models will be adequate for CAD/CAM applications[3].

1.7 SYSTEM DATA AND DATABASE REQUIREMENTS

Any system that offers CAD and CAM features must be able to perform certain basic tasks listed below. Implementation of each task should be done with due consideration to the other tasks in the system. The implications of each action on the remainder of the system should be analyzed carefully before implementation.

1. Part Design

2. Low-level data manipulation operations such as retrieve, insert, maintain (delete and add), etc.

3. Geometric data manipulation: Finite element analysis, manufacturability evaluation and other engineering operations.

4. High-level geometric operations such as: rotate, modify, replicate, etc.

5. Manufacturing operations: Define a blank, take a machining cut on the blank, generate numerical control machine code and generate process plan.

Clearly, data in a manufacturing environment is used by a host of people. The database and associated database management system's requirements are:

1. Data availability: every user should have access to a certain domain of the database. Ownership and group privileges are given to data members. For example, the modification of part geometry is done only by the designer, but the manufacturing engineer needs to be able to read the data. The manufacturing engineer may make a few changes to some data values of the part but does not have privileges to redesign the entire part.

2. Data integrity: The database management system should be able to ability to check for data consistency. When handling a variety of data objects, builtin error handling routines perform type and instance checks and determine if data is within the specified range of the system. The ongoing functions and operations of the system should not corrupt the data.

3. Data security: This is closely associated with (1) where data usage is partitioned functionally. In a large database, data objects are subject to operations by classes of users. The domains of usage are specified to allow access of certain data to certain users in a specific manner. Consider this small example of data access:

Operation:	Part Design.
Active attributes:	Geometric data, material and tolerances.
Owner Group(s):	Design (read and write access)
User Group(s):	1. Manufacturer (read access Geometric data, read/write access for tolerances)
	2. Cost Estimator (read access only)

4. Data association: Most operations require data to be associated with some function. Operations on data elements are performed in a manner most efficient for data retrieval purposes. In such applications, it is common to group data elements together. An associativity function between data elements and their operations is formally defined. Such grouping of data objects is commonly found as a language construct in object oriented programming.
   In a repetitive manufacturing environment, there exist several data objects that are closely related to each other. Some of their attributes are identical whereas other attributes differ. In such circumstances, it is an efficient means to define data as objects that have relationships. For instance, many mechanical assemblies have common components. It is efficient to have a data object called gearbox and have instances of a gearbox.

5. User Interface: A CIM system should be capable of defining data objects and operations in a user-friendly manner. For design purposes, the user must be able to specify a part in terms of volumes or blocks and other forms that have higher level descriptions than basic points or lines. Using these system specified primitives, the designer can

create complex components.

6. Error Checks: An error free environment is most important to the stability of the system. At every stage of design and manufacture, error checks are built in to ensure the integrity of data and related operations. A designer can erroneously assemble two components or a manufacturing engineer can perform an operation damaging the fixture.

Having defined broadly what integrated systems should accomplish, the scope of this study will be outlined.

1.8 PAPER OVERVIEW

The purpose of this thesis is to precisely define the basic data elements and their functions applied to a CAD/CAM system. The mathematical formulation of various functions and operations in a feature-based CAD/CAM system have been developed. Semantic descriptions of basic data elements and their associated operations are also developed. Finally an example computer implementation following these guidelines is presented.

The approach to the development of this system is designing with features, as opposed to a features extraction system. A set of semantic rules and functions to govern the set of orthohedral features (where planes are oriented at

right angles to each other) has been defined and developed. This set of features does not specify a system for all possible geometry. A subset of prismatic geometry has been chosen.

Mathematical representations of these rules are presented in the next chapter. The semantic equations represent a typical terminal session of a user in a feature based CAD environment. Flow of operations, hierarchical structuring and error checks are exhaustively covered by these equations. A set of manufacturing functions are also described.

The semantic structuring is implementation independent; only input-output specifications of each operation or function are described. The implementation of a part of the semantically designed system with input-output operations following the rules established are explained in Chapter 4.

This project has provided insights to problems encountered by users and designers of part design systems. Efforts have been made to alleviate these problems using the object oriented programming approach (a short introduction to object oriented programming practices is included in the Appendix) to feature based design. A general purpose step has been taken toward general purpose part description. Implementation of the concepts has been attempted.

2. DESIGN SEMANTICS

This chapter will cover design considerations and operations for computer representation of orthohederal prismatic geometry. Denotational semantics are used to define the major system operations of CAD systems in this domain. From the system designer's point of view, a CAD system should cover the areas of geometry creation, database management and geometry manipulation (including graphic displays, linear and angular transformations). The CAD system should also describe design data in a manner which is directly usable by manufacturing engineering.

2.1 METHODOLOGY OF APPROACH

A host of methods such as flowcharts, English-like definitions of methods, etc., are available to design complex systems. The approach taken to design this system is denotational semantics. Denotational semantics has been used to give mathematical meaning to programming languages and systems [7]. A semantic definition is valuable to implementors and programmers for the following reasons:

1. A precise standard for a computer implementation is set and this guarantees the same implementation on all machines.

2. The semantic code is useful documentation for a programmer to read formal semantics definitions and use as reference.

3. It is a tool for elegant implementations.

4. It guarantees correct implementation of the system. Semantic definitions are very similar to actual implementations.

The design modules presented later in this chapter ensure a standard output given the same specified input, regardless of the implementation. The semantics (defined as giving meanings to sentences) are syntax and language independent. A broad overview of the entire design process is described.

The syntax and notations in this paper are standard denotational semantics [7]. A LISP-like language syntax and structure for defining logic and data structures have been used.

2.2 SYSTEM DOMAIN

The semantics developed are low-level to the system and require mapping in order to make it convenient to use. Typically, the system in this chapter will be the kernel and application programs mapping user functions to the kernel functions will complete the link from a user to the system.

2.2.1 Geometric Domain

The geometric domain of this system is orthohederal geometry (planes at right angles to one another). In other words, the system operating domain is rectangular blocks. To be consistent with design, the effect of manufacturing operations is mapped to the geometric domain. This constrains manufacturing operations to be only of the type that will remove a rectangular block of material and thus the orthohederal property is retained for all rules and operations.

The basic set of system primitives is a set which includes all rectangular components. The related set of features includes geometry described by rectangular blocks. Any design or manufacturing operation on a part is reduced to a canonical form of description (rectangular blocks) as input to the basic system.

2.2.3 Denotational Semantics Syntax

It is intended here to develop a syntax to construct semantic sentences for various operations. The following short primer is standard denotational semantics syntax defining relationships between domains and co-domains.

1. Functionality operator for product domains.

   f: D1 X D2 X D3 X .....Dn -> A

   For an operation f, it's functionality says that f needs one argument from domain D1, one from D2,.....and one from Dn. to produce an answer in domain A.

2. Functionality operator for sum domains.

   f: (D1 + D2) X (D3 + D4) x ....4 -> A

   For an operation f, it's functionality says that f needs one argument from domain D1 or domain D2, one from domain D3 or domain D4, ... and so on. Thus the plus operator or the sum domain specifies an argument to be from the disjoint union from the domains operated upon by the plus operator.

3. Extensionality of a function and the assembly principle

   The form (α(a,b).c) is an element in A -> B. Alpha has been used for notational convenience in place of the usual lambda. The alpha function is called an abstraction and

defines the function f: A -> B as $\alpha(x1, x2, x3, .., xn)$. where $x1, x2, x3, \ldots\ldots \epsilon A$.

4. Other syntactic notations.

The remainder of syntax used during the development of the semantics to follow are common list processing (LISP) syntax. The comma in particular denotes a concatenation of lists or English-like "and".

2.2.2 Operational Domain

The domain of operations has been carefully chosen for retaining the orthohederal properties of the components. This means that rotational transformations about any axis are not defined. Rotation about the coordinate axes can be done only by ninety degrees. In the domain of manufacturing operations a machining cut about any arbitrary axis is not defined. Only cuts which are parallel to one of the edges of the component are allowed.

2.3 DATA PROCESSING OPERATIONS

The basic data structures are generalized lists. This is because of the general nature of lists implementation in any programming language. Generalized lists are readily available as language constructs in many languages and can be implemented with ease in others. The various operations that are available in lists are explained.

For design operations, fourteen basic and derived data-types have been used. The environment datatype encompasses all the data in the design and manufacturing system. The environment datatype is a compound list containing three distinct lists. They are the block-list, compound-list and tolerance-list. All other basic and derived data belong to these lists (refer to Figures 2.1 and 2.2).

2.4 BASIC DATA TYPES

Basic data types for manipulation of higher data are defined with semantic operations. Each definition follows a standard syntax. The first line specifies the datatype, the second line the domain and the range. All additional lines specify operations in the domain of the datatype.

Abstract Syntax and Semantic Algebra:

I. Real numbers

Domain r ϵ Real

Operations

+,-,*,/ -> Real

=,>,<,>=,<=,<> -> Real

Compound list

Comp-id-1	comp-id	block-id	block-id
Comp-id-2	block-id	comp-id	comp-id
Comp-id-3	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.

Block -list

Block-id 1	l	w	h	x	y	z	angle 1	angle 2	angle 3
Block-id 2									
.									
.									

Tolerance list

Tolerance 1	Face 1	Face 2	U tol	L tol
Tolerance 2				
Tolerance 3				

FIGURE 2.1 OVERALL DATA STRUCTURE

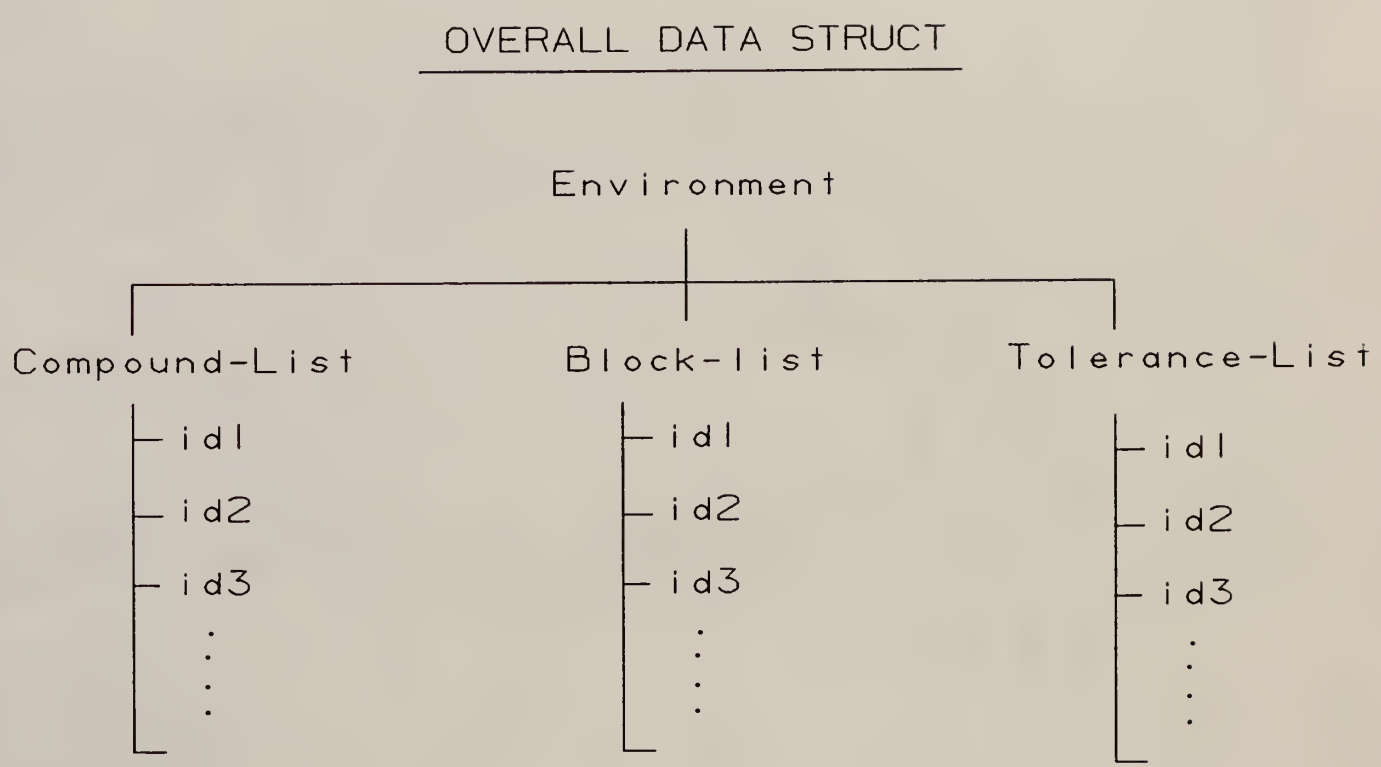


FIGURE 2.2 DATA GROUPS IN ENVIRONMENT

II. Natural Numbers

Domain n ϵ Nat

Operations

zero, one, two .... -> Nat

III. Truth Values

Domain t ϵ Tr

Operations

true, false -> Tr

IV. String

Domain ϵ string

Operations

A, B, C, D, E ... : String

2.4.1 Lists

Lists are grouped atomic data. A list has an infinitely increasing size that is allocated dynamically. The following definitions are for generalized lists.

V. Lists semantics

Domain l ϵ List

Operations

car : List -> Atom returns atom from the first position of list

cdr: List -> List returns list from the second position

append : list X List X ... -> List

append = α(l1, l2, l3, ...). concatenates lists

cons : Atom X List -> List

cons = α(a, l). concatenates an atom to a list

tail: List X Nat -> List

tail = α(l,n). returns list from nth position

length : List -> Nat

length = α(l). returns length of a list

ref : List X Nat -> Atom

ref = α(l, n). returns nth atom of list

make-list : Atom X Atom X Atom .... -> List

make-list = α(s1, s2, ...). creates a list

delete-fr-list : Atom, List -> List

delete-fr-list = α(a, l). deletes atom from list

The basic datatypes are as defined above. Derivations of the basic datatypes will form higher types of data suitable for application.

2.5 APPLIED DATATYPES

For design, the next level of datatype is called "block". Blocks are identified by a label and other attributes which are combined to make a rectangular block. These blocks serve as primitives to this system. These attributes form a list of n atoms. A block in the proposed system has ten attributes. More attributes can be added at a later stage by the applications programmer to describe system primitives better. A list of blocks completes the block definitions and is called "block-list".

A list of block identifiers may be combined to form a "compound". Once one compound is defined, other compounds can be defined by the following combinations:

1. block + block + block + ...
2. block + compound + block + block + compound + ...

This is called overloading since "compound" can take either one or more elements from domain "block" or one or more elements from domain "compound" or combination.

Finally, a datatype called tolerance is composed of a pair of block face identifiers and the distance between the faces. Tolerance on a linear dimension is specified by two real numbers specifying the upper and lower tolerance. The linear dimension is the distance between two faces which limit the linear dimension. Thus the tolerance is specified on the distance between any two parallel faces. Detailed definitions are covered in the tolerances section (2.8).

All these basic datatypes form a user primitive called environment. A user can create an environment which is comprised of blocks, compounds and tolerances. The purpose behind the definition of the environment is that the state of the system is monitored at every stage and any changes to blocks, compounds or tolerances are reflected directly in the user environment.

2.6 BLOCK DATATYPE AND OPERATIONS

The datatype "Block" forms the next primitive domain of the system (Figure 2.3). A block is represented by:

Block(block-id, length, width, height, origin-x,
origin-y, origin-z, orient1, orient2, orient3);

where: block-id is the block identifier
length is the length of the block
width is the width of the block

height is the height of the block

origin-x is the x-coordinate

origin-y is the y-coordinate

origin-z is the z-coordinate

orient1, orient2 and orient3 give the Z-Y-Z Euler angles for block orientation.

The following details the basic block operations. More operations on blocks are covered in Section 2.9 where a list of blocks exist in an environment and operations change the environment attributes.

Abstract Syntax and Semantic Algebra:

VI. Block

Domain: b ϵ Block -> List

String X Real X Real X Real X Real X Real X Real X Real X Real X Real

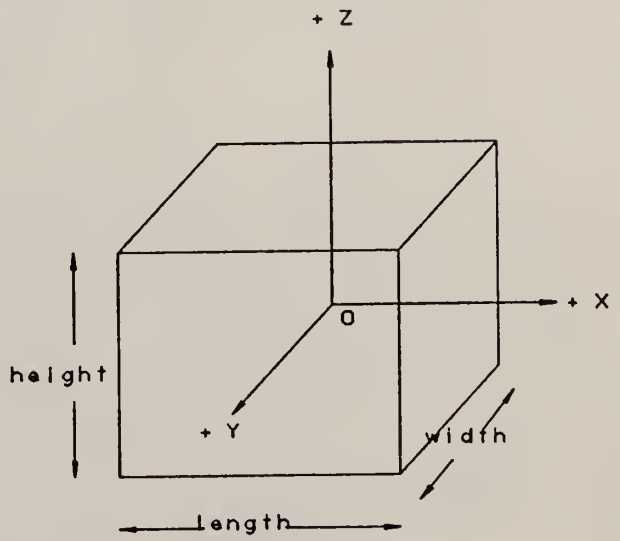
This operation constructs and defines a block in space with the ten attributes.

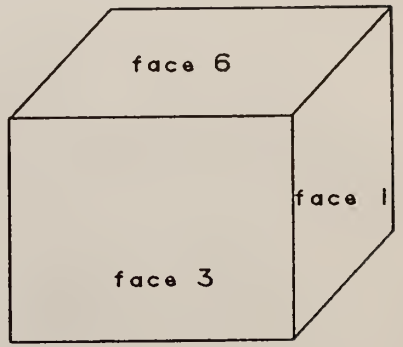
Operations

length-block : Block -> Real

length-block = α(b). ref(b,2)

This operation returns the length of the block.





Block Parameters

length, width
height, center coordinates,
angles of inclination
with respect to the
world coordinate system

Face Parameters

opposite face 1 -> face 2

opposite face 3 -> face 4

opposite face 6 -> face 5

FIGURE 2.3 BLOCK ATTRIBUTES

width-block : Block -> Real

width-block = α(b). ref(b,3)

Given a block, this operation returns the width of the block.

height-block : Block -> Real

height-block = α(b). ref(b,4)

origin-block : Block -> List

origin-block : α(b). make-list(ref(b,5),
ref(b,6), ref(b,7))

Operation to return the coordinates of the origin of the block. The center point (x/2,y/2 and z/2) are the local co-ordinate origins and x-origin, y-origin and z-origin are global coordinates with respect to the body or local coordinate system.

volume-block : Block

volume-block = α(b). length-block(b) * width-block(b) *
height-block(b)

operation to return the volume of a block.

display-block : String -> Image-List

display-block = α(b). display(b)

A display function mapping attributes of a block to a view port is assumed to exist - display(b). This function displays a block.

2.7 COMPOUND DATATYPE AND OPERATIONS

A compound body can exist in the two forms:

1. Block + Block + Block + ...

That is, the compound above is a combination of blocks with no other type of primitive domain. It has a single domain of type block. Note that prior to the construction of this type of compound, its block attributes must exist. Figure 2.4 gives a graphical example of the composition of a compound.

2. Block + Compound + Block + Block + ...

This type of primitive domain has blocks and compound attributes. The same rule of pre-existance of attribute domains must be satisfied.

In the actual implementation, a single type of primitive object has been defined. This takes advantage of language constructs in object oriented programming called overloading. The basic object (let us call this "solid") can take attributes corresponding to the type block or the type compound. Type matching is done and corresponding

values allotted accordingly.

The structure of a compound is a list of strings representing the compound domain's identification tags. The only isolated operation on a compound is a query function for its primitive domains. When a compound is queried for its components, it recursively concatenates all block identifiers into a list. Hence the query function returns a list of all blocks in a compound and their hierarchical relationships to higher compounds.

VII. Compound

Domain : c ϵ Compound -> List

Compound = (Block + Compound) X (Block X Compound) X

.....

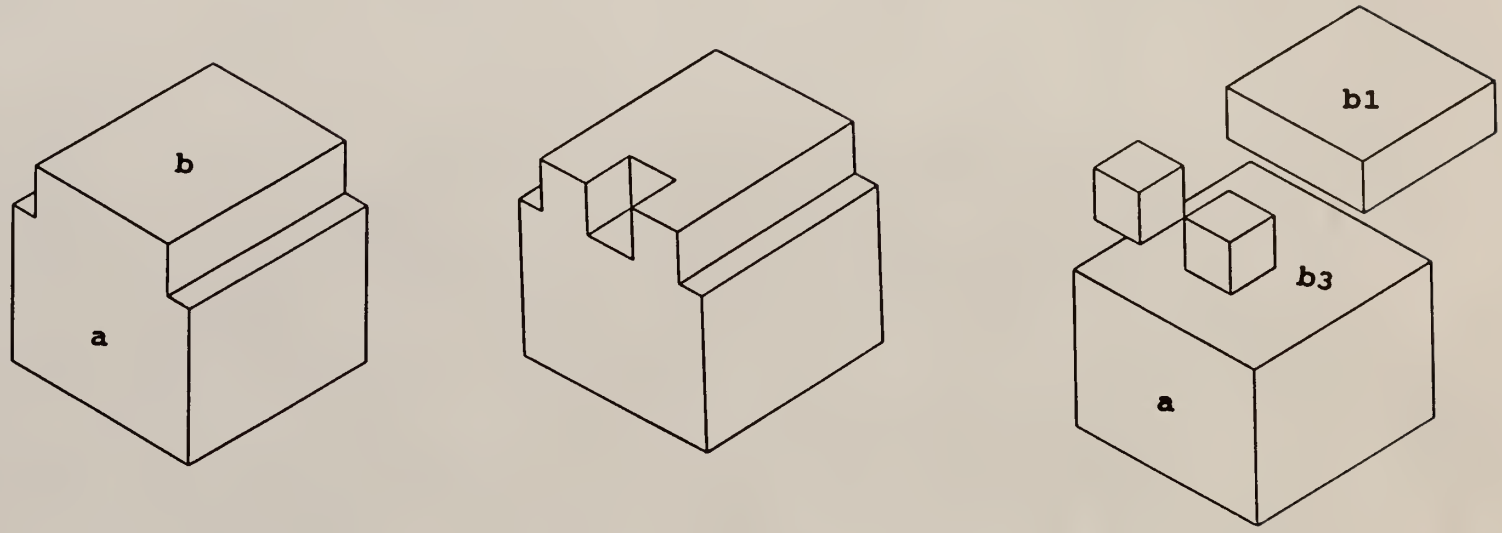
Compound is a disjoint union of Block and Compound. Built-in error checks ensure the creation of compounds that are comprised of domains which have already been defined.

Abstract Syntax and Semantic Algebra

compound : c ϵ Compound -> List

(Comp-id + Bl-id) X (Comp-id + Bl-id) X ..

compound = α(a). cases a of



a, pocket(b(face-6)) -> a, block(b1, b2, b3)

Alternately for the same geometry,

a, slot(b,face-3) -> a, block(b1, b2, b3)

FIGURE 2.4 COMPOUND BLOCK ATTRIBUTES

isBlock(a) -> cons(c, bl-id)

isCompound(a) -> cons(c, comp-id)

Constructs a compound.

Operations

comp-id-list : string -> list

comp-id-list = α(c). isCompound(c)

The input is a compound identifier. The function returns the identification tags of primitive domains' associated with the compound.

2.8 TOLERANCE DATATYPE

Tolerances in manufacturing situations are specified as parameters of dimensions. A tolerance in a manufacturing situation indicates the amount that a feature may deviate from its basic dimension and still function properly[12].

A general unsymmetrical bilateral tolerance is assumed. Bilateral tolerances are related to the basic dimensions in two directions. Unsymmetrical tolerances are those representing unequal tolerances in the two directions.

A general purpose notation has been adapted for the representation of tolerances. The tolerance on a single dimension bounded by two faces is specified by the two face-

ids and real numbers specifying the numeric values of the upper and lower tolerances. Thus any side, length or linear dimension of objects can be represented.

The datatype face is an attribute of tolerance which has to be defined prior to tolerance. Hence we introduce the datatype face which has the attributes of the form:

Abstract Syntax and Semantic Algebra

VIII Face

Face : f ϵ Face -> List

String X Nat

where

String is the block-id

Face is its face number

IX Tolerance

Tolerance : t ϵ Tolr -> List

Face X Face X Real X Real

where

Face - the two face attributes are the boundaries of the dimension whose tolerance is specified.

Real - the two numeric values of the dimensional tolerance. The convention is that the first Real is the upper tolerance

and the second Real is the lower tolerance. Further operations on tolerances are covered in the next section.

2.9 ENVIRONMENT OF THE CAD SYSTEM

This is the final primitive domain of the CAD system. Environment defines the system state at any time. This domain is closest to the user interface. In the overall system, the interfaces and user domain systems will map into the environment domain. The environment captures and stores information about the system state.

The primary attributes of the environment domain are:

1. A list containing the primitives called block-list. The block list contains the various blocks created in the system. The blocks created in the system are stored as lists. Refer to Figure 2.1 for the block list representation.

2. A list containing the names of compound primitives created since the system epoch. Referring to Figure-2.2, it becomes apparent that Comp-List is a list of lists containing identifiers. The identifiers represent the IDs of compounds created and the rows contain all objects in one compound.

3. Tolerance list is similar to a block-list and contains attributes of tolerances.

Thus, the notation of the environment can be written:

environment : e ϵ env

= block-list(e) X comp-list(e) X tolr-list(e)

where

block-list : List X Nat -> List

block-list = α(b,n). (b,n)

comp-list : List X Nat -> List

comp-list = α(c,n). (b,n)

tolr-list : List X Nat -> List

tolr-list = α(t,n). (t,n)

Abstract Syntax and Semantic Algebra

X. Environment

environment : e ϵ env

= block-list(e) X comp-list(e) X tolr-list(e) -> List

Operations

create-block : env X block -> env

create-block = α(e,b). ((append(block-list(e),b),
comp-list(e), tolr-list(e))

Creates a block in the environment and adds the block to the list of blocks.

block-exists : env X String -> String

block-exists : α(e, block-id).
car(car(block-list(e))) =
block-id -> block-id
else
block-exists(cdr(block-list),
comp-list(e), tolr-list(e),
block-id)
else
error -> error

This operation searches a list of blocks to see if block has been defined and is used for error checks. It returns the name of the block if the test is a success and an error code otherwise.

comp-exists : env X String -> String

comp-exists = α(e, comp-id). car(car(comp-list(e)) =
compid -> comp-id

```
                else
                    comp-exists(block-list(e),
            cdr(comp-list(e)), tolr-list(e), comp-id)
                else
                      error-> error
```

To check if a compound has been defined or not.

decode-block-id : b-id ϵ block-id

```
                env X String -> Block
decode-block-id = α(e,block-id).  block-exists(e,block-id)
                      = error -> error
                else
                  car(car(block-list(e)) =
                     car(block-list(e))
              else
                 decode-block-id(cdr(block-
                   list(e), compound-list(e),
                     tolr-list(e), block-id))
```

This function takes as input a block-id and return a block with the block parameters.

decode-comp-id : c-id ϵ comp-id

```
                env X string -> env
decode-comp-id : α(e, comp-id).
comp-exists(e,comp-id) =  error -> error
                car(car(comp-list(e))) = comp-id ->
```

```
                    car(comp-list(e)
                else
               decode-comp-id(block-list(e),
               cdr(comp-list(e)), tolr-list(e), comp-id)
```

This function returns all attribute values of a compound given its identifier. This function is not recursive. The next function comp-id-list extracts all atomic element information to list all elements in block form when queried.

```
comp-id-list : List X String -> comp-list
comp-id-list :  α(e, comp-id). f = decode-comp-id(e,
                          comp-id)
               comp-exists(car(cdr(f))) =
                   error -> { block-exists(car
                   (cdr(f))) = error -> error}
               else
                append(comp-list,car(cdr(f)))
               else
                   comp-id-list(cdr(comp-list(e),
                        car(cdr(f)))
```

This function recursively extracts a block from a compound given a comp-id.

```
modify-block : env X String X Nat X Real -> env
modify-block =    α(e, block-id, n, r).
```

```
                block-exists(e, block-id) =
                     errorvalue1 -> error1
                else
                     car(car(block-list(e))=
                          block-id -> [n|->r]
     car(block-list(e))cons (cdr block-list(e)),
           comp-list(e), tolr-list(e), block-id, n, r)
```

Above function modifies one attribute of the block at a time. The input parameters to this function are the block-id and the element number in the list to be modified (n). The new numeric value of the nth parameter in the list (r) is specified and replaced at the nth position. The semantic expression [n|->r] seeks the nth position in the list.

create-compound : env X (block-id + comp-id) X (block-id + comp-id) X String -> env

```
create-compound = α(e,a,b,comp-id).     comp-exists(comp-id)
                   <> error -> error
                   cases a of
                        isBlock(x)
                   block-exists(x) = error ->  error
                        isCompound(x)
                   comp-exists(x) = error ->  error
                   cases b of
                        isBlock(y)
```

```
                    block-exists(y) = error -> error
                         isCompound(y)
                    comp-exists(y) = error ->  error
                    block-list(e), append(comp-list(e),
                    make-list(comp-id,x,y)), tolr-list(e)
```

This function takes an attribute from the disjoint union of comp-id and block-id and takes the second attribute of the same form. The function then checks to see if comp-id is already defined and the other IDs defined or not. A compound is formed by appending identifiers to the list.

```
Display : display-list ϵ List
          (comp-id + block-id + compound) ->
                    display-list
Display : α(a). cases a of
                    isBlock(a)
    block-exists(block-id) = error -> error
                    { else display a }
    isCompound(c) -> { car c = nil ->
           null-display-list
         else append(display(car
          cdr(c))), display(cdr(cdr(c))) }
```

The display function is overloaded to take a block-id or comp-id and display the corresponding object. The "display" function mapping attributes of a block to a viewport is called recursively to display a block when extracted from

the comp-id. Output of this function is in the form of a list which keeps track of blocks displayed.

delete-block-comp : list X String X String -> list

```
delete-block-comp = α(e, block-id, comp-id).
          block-exists(e, block-id) = error ->
                    comp-exists(e, comp-id) = error
                         -> error
            else
               delete-fr-list(block-id,
                    decode-comp-id(comp-id))
```

This function deletes a block from a compound.

delete-comp-comp : List X String X String

```
delete-comp-comp =  α(e, comp-id1, comp-id2). comp-id1 =
          comp-id2 ->
     comp-exists(e, comp-id1) = error  ->
          comp-exists(e, comp-id2) =
                         error -> error
     else
        remove-fr-list(comp-id1, comp-id2)
```

This function removes a compound attribute from a compound. The function will not allow a compound to be deleted from itself. If both identifiers are the same, it returns an error message.

```
volume-comp : List X String -> Real
volume-comp =  α(e, comp-id). comp-exists(e, comp-id) =
                    error -> error
        else
      f = comp-id-list(e, comp-id)  ->
                volume-block(car(cdr(f))) +
               volume-comp(cdr(cdr(f)), car(
                        cdr(cdr(f)))
```

Recursive function to find the volume of a compound block.

```
translate : List X (String + String) X Real X Real X Real
                          -> List
translate =  α(e, a, r-x, r-y, r-z).  cases of
                    isBlock(b) ->
      modify-block(e, car(b), 5, r-x) ->
      modify-block(e, car(b), 6, r-y) ->
      modify-block(e, car(b), 7, r-z)
                    isCompound(c) ->
      translate(cdr(c), car(cdr(c), r-x
                     r-y, r-z)
```

```
add-to-comp : List X (block-id + comp-id) X  (block-id +
                comp-id) -> List
add-to-comp : α(e, a, b). cases a of
     comp-exists(e, car(a)) = error ->
```

```
        error
    else
      cases b of
    comp-exists(e, car(b)) = error ->
     block-exists(e, car(b)) =  error -> error
     else
      cons(a, car(b))
    else
      append(a, b)
```

This function checks to see if argument is a compound or block and appends it to the comp-id attributes, thus adding the block or compound to a compound.

write-to-disk : env X (String + String) -> file

```
write-to-disk = α(e, a). cases a of
                    isBlock(x) -> write(x)
                    isCompound(y) ->
                     write(comp-id-list(car(y))
```

Function to write a block or a compound to disk. If a block-id is supplied, writes attributes of block. If a compound-id is supplied, function writes corresponding identifiers and recursively the block parameters to disk.

2.10 EXAMPLE DESIGN CREATION

The following examples will demonstrate the use of the design semantics developed. A compound part will be created using three blocks- B1, B2 and B3. The resulting compound is COMP. The intermediate compound C1 is created by the combination of two blocks. COMP is finally formed when C1 is combined with the remaining block. The individual block parameters for B1, B2 and B3 are as follows (refer figure 2.5 in the design example).

```
create-block(B1, 4, 4, 4, 10, 0, 0)
create-block(B2, 1, 4, 1, 11.5, 0, 2.5)
create-block(B3, 1, 4, 1, 8.5, 0, 2.5)
```

The orientation parameters have been omitted because of orthohedral geometry. However to represent orthogonal rotations, the angles can be zero degrees, ninety degrees and ninety degrees (Figure 2.5).

Constructing a compound with these parameters, the semantic relations are:

```
create-compound(e, B1, B2 , C1)
create-compound(e, B3, C1, COMP)
```

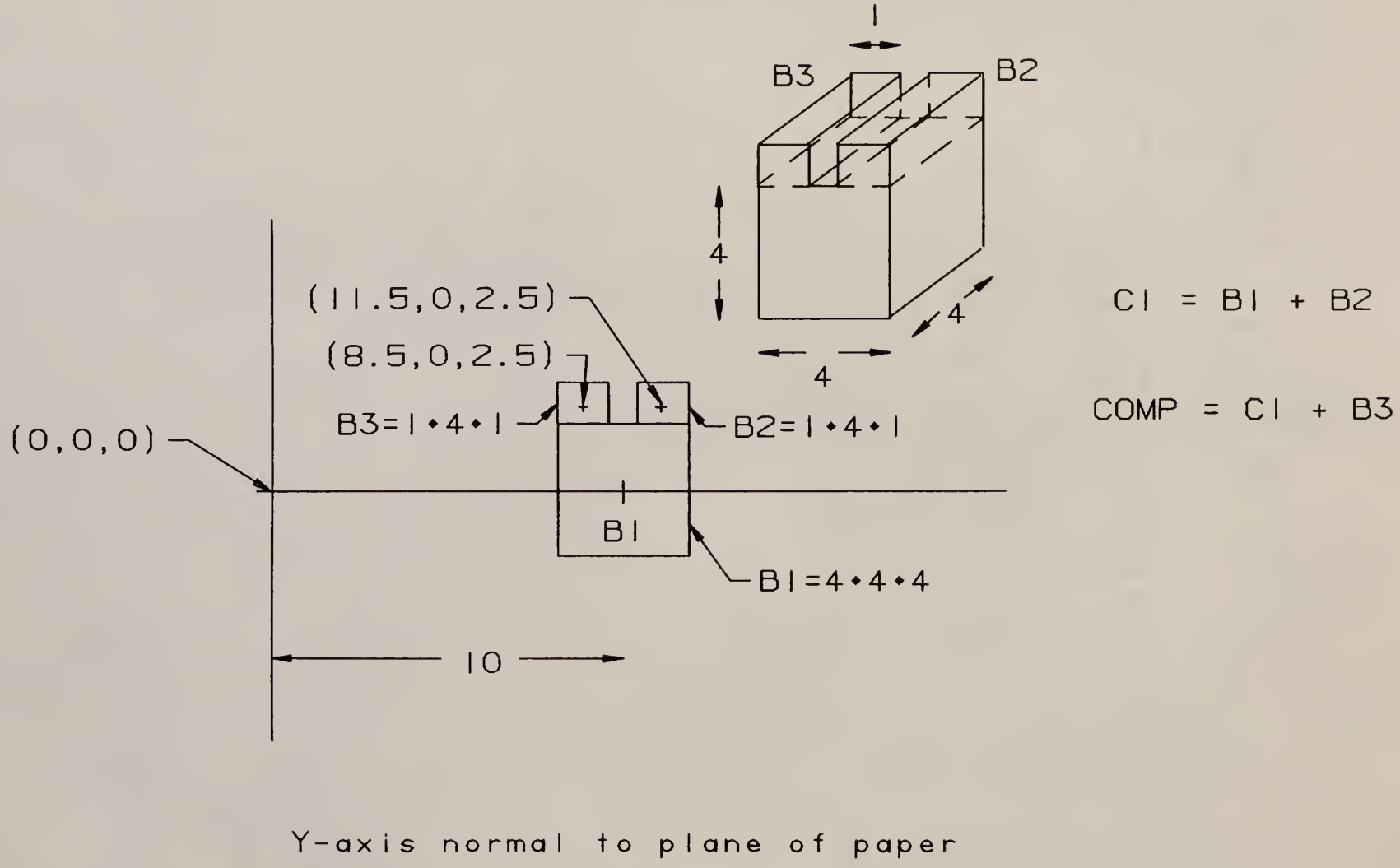


FIGURE 2.5 EXAMPLE DESIGN

Using the compounds C1 and COMP just created, a user may query the compounds, delete a block from the compounds or create more blocks and add to the compound. As mentioned earlier, error checks for interferences are assumed to exist and only integral data filtered down to these low-level functions.

2.11 SUMMARY OF THE METHODS DERIVED

The design operations would become the heart of a CAD system for orthohederal geometry. These are low level functions and definitions. Mapping to include user interfaces is recommended before full implementation.

An object oriented programming environment would be ideal to represent and effectively communicate the primitive methods and definitions described earlier in the chapter. When combined with object oriented databases, it could become a powerful tool for effective CAD in the domain.

The applications development for the functions presented would involve central software development and application programming. In later chapters a few implementation examples will be presented. Methods to map the low level functions to user commands will be described and it will be shown how language constructs such as data

abstraction are essential for developing a CAD system of this nature. The output parameters are language and machine independant which ensures a constant output.

Note: The greek alphabet alpha has been used as a notation in the semantic functions described in this chapter instead of the usual notation of lambda.

3. MANUFACTURING SEMANTICS AND RELATIONSHIPS TO THE CAD SYSTEM

This chapter defines various manufacturing operations and their relationships to the proposed design system. An effective CAD and CAM system should be developed with consideration for problems to be shared by the component systems. A compromise should be sought between the constraints of the CAD and CAM parts of the system.

The manufacturing functions and their relationships to the part description are not as straightforward as design functions. Design features are easier to model than are manufacturing features. This is due to the nature of the data to be represented. In CAD, the main goal is to represent a part by its geometric attributes and to manipulate them in order to achieve the topology of a part previously described. Most CAD systems allow the designer to use previously defined geometry (CAD libraries) to build the part and provide functions with which to manipulate geometric data (rotate, translate, etc).

Manufacturing functions, on the other hand, manipulate abstract, amorphous data. There are a myriad of attributes to manufacturing data, including CAD data. The effect of a manufacturing operation on a blank has to be represented in concrete terms to attain the goal of automating CAD and CAM. Representation of manufacturing functions is by the inference of the action of a tool on the blank. When a tool operates on a blank, the questions that arise are: What is the kind of cut? What will be the effect of the cut on the workpiece? Where is the workpiece and the location of the machining cut? How should the cut be represented? These are typical of the problems encountered in manufacturing. It is not easy to answer them all at once. The action of machining operations has to be represented in concrete terms. This form of representation is a pressing need in order to acheive automatic production process planning for generic components.

The approach taken to represent manufacturing data in this research effort is to convert the manufacturing operations to CAD data representing the blank. During a machining operation to cut a blank, the manufacturing operation (eg. cutting) is converted to a function of the blank geometrically -- thus manufacturing operations reflect directly on CAD data. There exist other attributes of

manufacturing operations such as tooling, cutting rate, etc. Extensions and additions will complete the representation of a component in terms of its manufacturing needs. This study has concentrated on the direct transformation of the machining operations to the CAD data of the workpeice.

Representation methods in manufacturing are currently being researched for defining a complete set of data elements to represent the elements of manufacturing[3],[8],[9]. One approach is to define all manufacturing machining operations as subtractive geometry. Another method is to keep manufacturing data separate from the design data; design and manufacturing data form separate inputs to a CAM system.

Subtractive geometry representation has a shortcoming -- dynamic representations of objects are difficult to obtain. In order to completely represent a component, boolean mathematics is performed. The difference of additive and subtractive geometry is the result. Updating data is a common problem -- boolean arithmetic can be performed on geometric objects, but what happens to the manufacturing data? The attributes of machining associated with subtractive geometry cannot be updated periodically. In subtractive geometry, it becomes especially difficult to represent a machining cut over a surface that is already a

function of some other machining operation. There is a need for a general purpose methodology representing manufacturing operations on generic designs with output data describing the state of the blank (in terms of design and manufacture).

Features extraction systems contain design and manufacturing data in separate formats. In such systems, manufacturing attributes enter the system after design is complete. Results from ongoing features research projects are yet not conclusive to measure efficiency in data representation.

3.1 REPRESENTATION SCHEME

In the domain of geometry defined in Chapter 2 the manufacturing operations will be defined additively to the geometric representation. Manufacturing operations will be added to existing lists of attributes to parts. A list of manufacturing operations and their parameters are stored for the purpose of undoing operations previously defined.

After the geometric description of a part or a blank, a user may decide to machine a single face. He would proceed to give machining cuts in increments, depending on the amount of material to be removed. If at any stage of operation he decides to "add material" or undo one or

several operations, system utilities to aid such supporting operations to manufacturing should be made available.

3.2 MAPPING MANUFACTURING OPERATIONS TO DESIGN

The prime goal of a manufacturing function is to reflect a manufacturing operation on the design data. After a blank has been defined and exists in the CAD database, a manufacturing operation should map onto the CAD database and change parameters of the blank representation. Thus, a computer emulation of manufacturing is the end result of a machining operation on a computer. The benefits from such an implementation are evident -- manufacturing practices can be perfected before actual machine tool operations.

The philosophy of the proposed mapping mechanism is:

1. For every manufacturing operation, determine the primitive domain operated upon.

2. Transform the result of the machining operation to alter the representation of the blank. Starting with a block, a slab cut changes the attributes of the block after the operation. A corresponding change should be made to the blank in the CAD data to reflect the manufacturing operation.

3. Store the manufacturing operation's attributes in a location for future reference.

4. Transform the blank definitions in the CAD database.

5. Transform a block definition to a compound if the manufacturing operation is going to change the overall geometry of the primitive domain.

6. Save copies of the old definitions to assist in undo operations (a specific number of undo operations can be specified due to computer space constraints considerable space can be occupied by repetitive manufacturing operations).

7. Reflect these changes in the environment.

3.3 DATA CONSIDERATIONS IN MANUFACTURING

New datatypes and operations to assist in transforming manufacturing operations to geometric entities are defined. Face definitions are extensively used since all manufacturing operations are defined with respect to faces in blocks and compounds. The face numbering system followed during the development of the methods is:

1. Face 1: The face perpendicular to the +x axis of the local coordinate origin.
2. Face 2: perpendicular to -x axis
3. Face 3: perpendicular to +y axis
4. Face 4: perpendicular to -y axis
5. Face 5: perpendicular to +z axis
6. Face 6: perpendicular to -z axis

The CAD system is used to define and extract blank data. The environment variables are not changed during manufacturing operations. Temporary copies of data are made and operated upon. When prompted by the user to save new data into the existing database, the old data is overwritten to store the new modified data as a result of manufacturing operations.

3.4 OPERATIONS IN MANUFACTURING OF FEATURES DOMAIN

To remain within the bounds of our CAD domain described, the major operations defined in this chapter must retain the orthohedral properties of the created parts. Thus the major operations in manufacturing the components described by the CAD system defined are:

1. Casting Allowances - After designing a component for its final dimensions, a casting allowance function

transforms the geometric data of a block or compound to represent a cast part. Input parameters of this function are the block or compound identifier, casting allowance and the face identifier. The last parameter ensures variable casting allowances on the different sides or faces of the block or compound. A new block or compound with an extension "cast" to the original identifier is formed.

2. Blank Allowances - These function to form an envelope of material in order to represent the block or compound as a blank (uniform rectangular cross section) to be machined. Adds blank allowance to a block uniformly on all the six faces. On a compound, the limiting dimensions in all three directions are determined and a blank allowance is allocated accordingly; a new blank in the form of a block with an identifier extension "blank" is formed.

3. Slab Cutting - A face is machined uniformly with the same thickness of cut on a single face. This operation is usually performed using a milling machine.

4. Slot or Groove Cutting - A face is machined to form a linear slot parallel to an edge contained in the face. A slot starts at one of the edges and ends at the

opposite parallel edge. A single instance of a slot is of uniform thickness and depth. Additional operations will transform the block to the desired shape.

5. The final manufacturing operation defined in our system is a rectangular pocket. Internal pockets are manufacturing features which are enclosed by material in a block. However, a pocket may intersect a block boundary and thus open pockets may be formed (see Figure 3.6). These cases are discussed.

3.5 CREATING A CAST PART

The designed part is given casting allowances with this function. Input parameters to the Cast-Part function are block or compound, face identifier and a casting allowance "call". Note that when the input is a compound and when a face is specified, only the block corresponding to the face-id specified is transformed to a casting with allowances on the specified face. The output parameter is a new block or compound corresponding to the input part type. This has the identifier "cast" as an extension to the old identifier.

Abstract Syntax and Semantic Algebra:

```
Cast-Part :  env X (Block X Compound) X Real X Face ->  env
Cast-Part =    α(e, a, call, f).  cases of a
      isBlock(a)
        car(a) <> ".cast"  ->  copy-block(e, a, append(
                    car(a), ".cast"))
      isCompound(a)
       car(f) <> ".cast" -> copy-block(e, decode-block-
              id(e, car(f))), append(
                    car(f), ".cast"))) ->
                         car(a) <> ".cast" ->{
       copy-comp(e, car(a), append(car(a),".cast")) ->
        delete-from-comp(e,car(a)".cast", car(f))   ->
     add-to-comp(e,car(a)".cast", car(f)".cast") }
                    -->
     cases of (f)
    isNat(x)  -> { x=1 ->
     modify-block(e,car(f)".cast",5,(ref(car(f)
     ".cast",5) + call/2))   ->
          modify-block(e,car(f)".cast",2,(ref(car(f)
                    ".cast",2) + call))
               else
          x=2  ->
```

```
 modify-block(e, car(f)".cast",5,(ref(car(f)
 ".cast",5) - call/2))   ->
 modify-block(e,car(f)".cast",2,(ref(car(f)
 ".cast",2) + call))
else
       x=3
      .........
      .........
else
x=4
.......
x=5 .........
x=6 ............
```

Semantics for different faces of x corresponding to the face numbers can be derived in a similar fashion. Each pair of the modify-block function modifies the block corresponding to the face identifier specified with a casting allowance - call.

3.6 SPECIFYING A BLANK SIZE

The input parameters to this function are block or compound identifier, a blank allowance "blall". Function creates a new compound with overall dimensions the limiting size of the block or compound.

Abstract Syntax and Semantic Algebra:

Blank-Part : env X (Block X Compound) X Real -> env

```
Blank-part =  α(e, a, blall).
     cases of (a)
      isBlock(a) { car(a) <> ".blank" ->
      copy-block(e,a,append(car(a), ".blank"))
      else
       modify-block(e,car(f)".blank",2,
       (ref(car(f)".blank",2) + blall))
                    ->
       modify-block(e,car(f)".blank",3,
              (ref(car(f)".blank",3) + blall))
                 ->
       modify-block(e,car(f)".blank",4,
          (ref(car(f)".blank",4) + blall))}
     isCompound(a)
      car(a) <> ".blank" -> copy-comp(e,car(a),
         append(car(a),".blank")) ->
      create-block(append(car(a)
        ".blank", ()))  -> create-block(append(car(a)
        ".blank"), car(find-max(a),car(cdr
              (find-max(a))),car(cdr(find-max(a)))))
```

This function finds the maximum parameters of an element from the disjoint union of a block and a compound. The

operation adds material and creates a new blank block. The function find-max recursively finds the limiting linear parameters of a compound.

3.7 SLAB CUTTING

Slab cutting of a block is the machining of a face to a uniform depth. Given a depth of slab cut, the cut can be effected by incremental depths or by a single cut depending on the depth of cut and machine parameters. The effect of a slab cut on a block face is to change one dimension in the block definitions. Slab cuts are defined to be parallel to the face being cut (Figure 3.1).

End milling and face milling with rotary cutters are two methods of slab cutting, this depends on the machine tool and tool attachments available. Generally, a face mill rotary cutter is a feature available on a universal milling machine and gives a faster rate of cutting and better quality of surface finish.

3.7.1 Semantics of a slab-cut

Slab-Cut : s-c ϵ slab

= s-c -> Real

Slab-Cut-Block : env X Face X Slab-Cut X Surface-Finish

X Tolerance -> env

Slab-Cut-Block = α(e,f, s-c,surfin,tolr). cases of

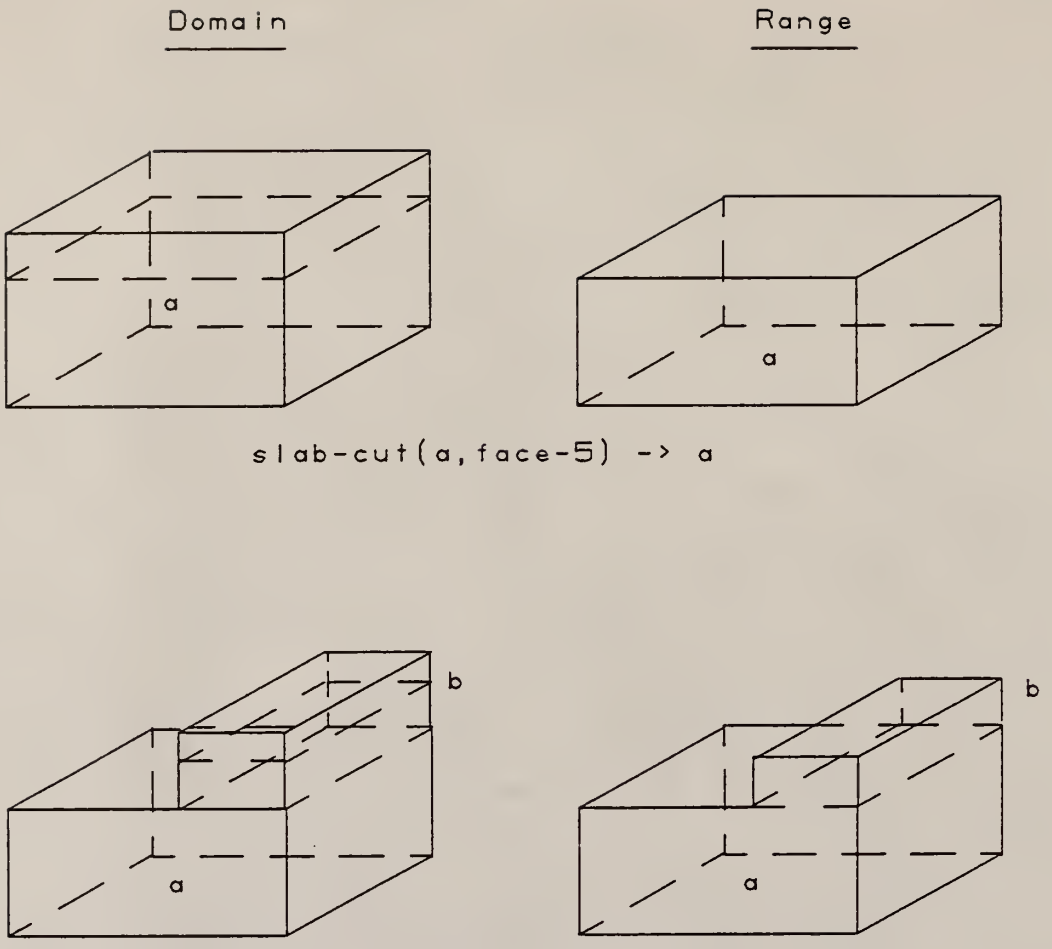


FIGURE 3.1 SLAB CUTTING TRANSFORMATIONS

```
cdr(f) isNat(x)  -> { x=1  ->
                  modify-block(e, car(f), 5,
              (ref(car(f), 5) - s-c/2))   ->
    modify-block(e, car(f), 2,(ref(car(f), 2) - s-c))
    modify-block(e,car(f),11,surfin) -> append(tolr-list,
tolr)
   else
     x=2  ->
    modify-block(e, car(f), 5, (ref(car(f), 5) + s-c/2))
                ->
    modify-block(e, car(f), 2, (ref(car(f), 2) - s-c))
modify-block(e,car(f),11,surfin) -> append(tolr-list, tolr)
    ..... }
```

In each of the pairs of the modify-block functions, the first function call changes the local coordinates of the block to center it. The next function call changes the parameters of the block's length, width and height depending on the face operated on by the Slab-Cut-Block function. The semantics can be repeated for all six faces of the block.

3.8 SLOT CUTTING OPERATION AND VARIATIONS

The slot cutting operation is one in which a block volume forming a subset of the parent is removed from a face. The constraint in the slot cutting operation is that the slot must be parallel to the edges of the block faces.

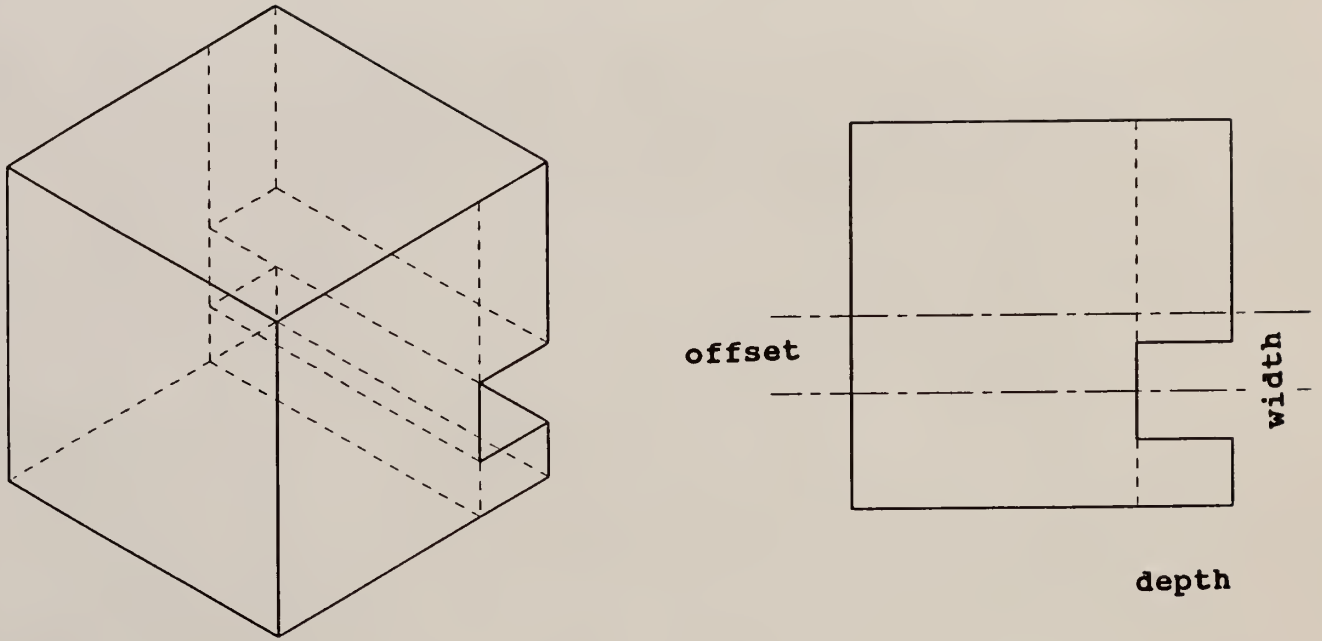


FIGURE 3.2 SLOTTED FACE CROSS SECTION

The manufacturing methods of a slot are end mill cutting, side and face rotary milling and are usually done with a horizontol or vertical spindle milling machine. Variations of slots (stepped slots are possible, the block from which the slotting operation must be specified) (Figure 3.2).

The following cases exist in slotting operations:

1. Slot centered in a face.
2. Slot offset in a face.

The semantic equations have been written for the general case of slot either offset or centered.

The mechanism of converting the slotting operation to the CAD representation is as follows:

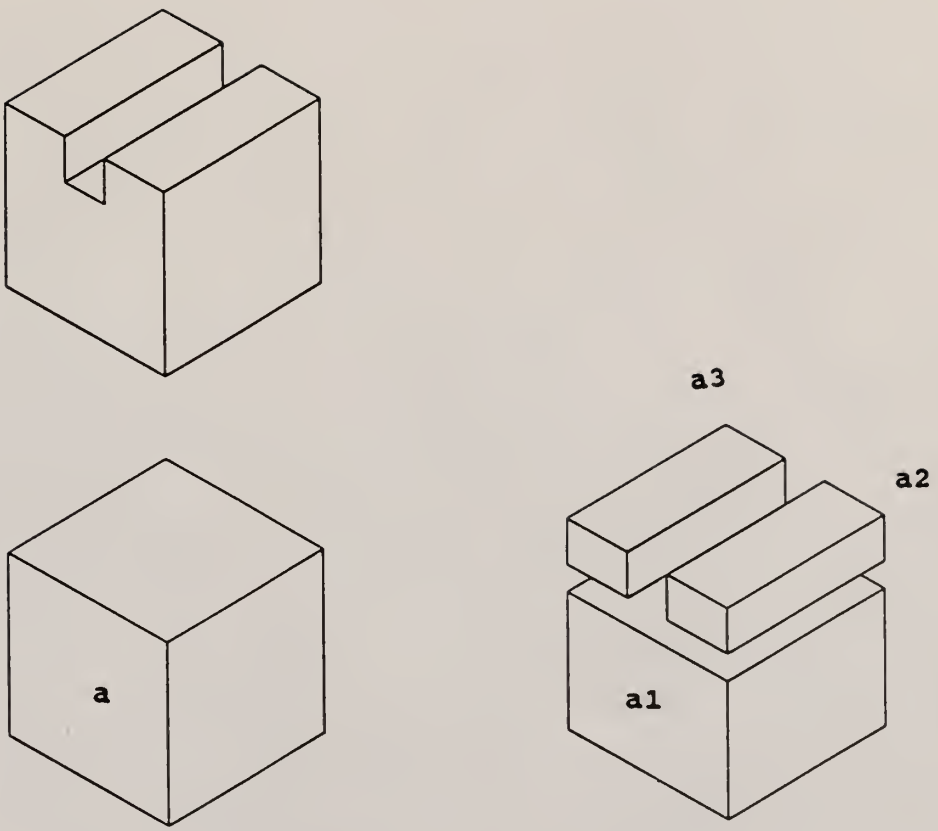
1. The face of the block being operated upon is first reduced to a block of the largest dimensions that excludes the slot. The slab milling operation with a depth equal to the slot depth reduces the original block.

2. The remaining blocks that represent the final geometry are constructed separately with parameters derived from the operation.

3. The base block from 1 and the blocks from 2 are are combined to form a compound.

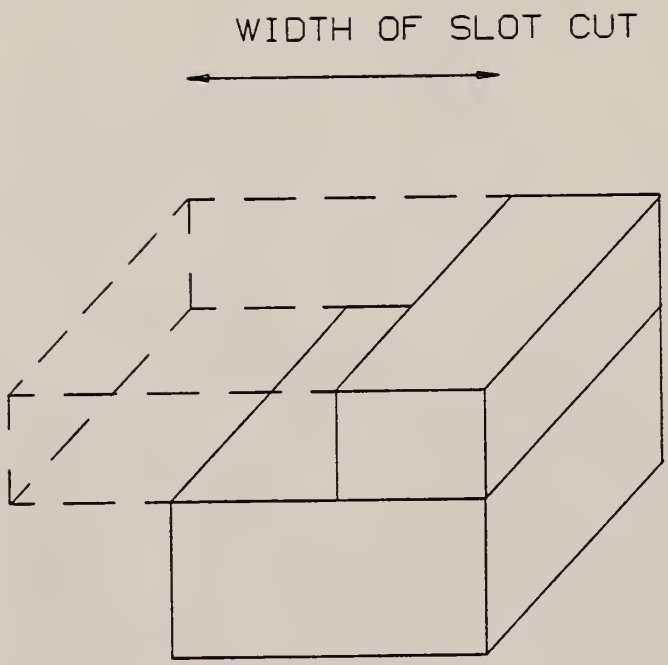
Thus, the geometry formed from the slot cutting operation performed on a block is represented in terms of a compound. The resulting representation is stored in the environment variables (Figure 3.3).

Variations of the slot cut exist that may result in unexpected geometry. Normally a slotting operation on a block causes three primitive blocks to be formed. If a slot is specified with unusual parameters, only two or even one block is likely to be formed (Figure 3.4). The recommended recovery for this kind of error is to have a cleanup operation after the slotting operation. The cleanup operation will delete entities formed with negative values of the length, width, height attributes of a block due to the operation. Thus the integrity of the database will not be affected after several operations.



slot(a) -> (a1,a2,a3)

FIGURE 3.3 GEOMETRIC COMBINATIONS DUE TO SLOTS



Exception of a Slot Cut - Specifications of slot-cut transforms block to unexpected geometry

FIGURE 3.4 VARIATION IN SLOT OPERATIONS

3.8.1 Slot Definition and operations

```
Slot-Cut-Block :  env X Face X Width X Depth X Tolerance X
                  Tolerance X Orientation
Slot-Cut-Block =   α(e, f, wid, dep, o, w-tolr, d-tolr).
                    cases of cdr(f)
    isNat(x)  ->  (
      copy-block(car(f).[1], car(f).old) ->
       rename-block(car(f).[1], car(f)) ->
        Slab-Cut-Block(f, dep) -> ( x=1 ->
     ( car(o) = yaxis ->
     create-block(e, car(f).[2], dep,
        ref(f,3), (ref(f,4)/2 + cdr(o) - wid/2),
      (ref(f,5) + ref(f,2)/2 + dep/2), ref(f,6),
    (ref(f,7) + cdr(o)/2 - wid/4 - ref(f,4)/4)))

     -> create-block(e, car(f).[3], dep,
    ref(f,3), (ref(f,4)/2 -cdr(o) -wid/2),
     (ref(f,5) + ref(f,2)/2 + dep/2), ref(f,6),
   (ref(f,7) + ref(f,4)/2, cdr(o)/2 - 3*wid/4))
      -->
   append(append(w-tolr,append(car(f).[2],6),
     append(car(f).[3],1)), tolr-list))
         -->
   append(append(d-tolr,append(car(f).[2],1),
     append(car(f).[2],2)), tolr-list)
```

The first set of equations define the operations of a slot cut on the face 1 of a block. Copy-block and rename-block operations initialize the data set. The slab-cut operation performs the first reduction on the block denoted by car(f).[1]. The subsequent create-block operations create the two blocks car(f).[2] and car(f).[3]. Append operations append the tolerances specified by the cut, the depth and width tolerance, to the corresponding faces first. They are then appended to the tolerance list in the environment.

The variable named "orientation" specifies the parallel axis of the slot and the offset. Orientation can be defined as:

Orientation : o ϵ orient

Orientation : String X Real

The string in orientation may be x-axis, y-axis or z-axis to specify the parallel axis of the slot. The real number denotes the offset of the slot. It must be a signed real number specifying the direction of the slot as well as the distance from the face center.

The semantics for a slot on face 1 parallel to the y-axis was given. Now the semantics for a slot in face 1

and parallel to the z-axis are as follows. The same logic as the previous case holds.

```
car(o) = z-axis  ->
 create-block(e, car(f).[2], dep, (ref(f,3)/2
+ cdr(o) - wid/2), ref(f,4), (ref(f,5) +
 + ref(f,2)/2 + dep/2), (ref(f,6) + cdr(o)/2
 - wid/4 - ref(f,3)/2), ref(f,7)) ->

create-block(e, car(f).[3], dep, (ref(f,3)/2
- cdr(o) -wid/2), ref(f,4), (ref(f,5) +
 + ref(f,2)/2 + dep/2), (ref(f,6) + cdr(o)/2
 -3*wid/4 - ref(f,3)/2), ref(f,7))
```

Create-block is a function defined in the design semantics section. It is called here with derived parameters to construct the respective blocks with formal arguments defined earlier.

Finally, the create-compound operation is called to create the compound consisting of car(f).[1], car(f).[2] and car(f).[3]. This compound is given the original identifier car(f).

3.9 POCKET DEFINITIONS AND OPERATIONS

An internal pocket is a manufacturing entity enclosed in a block. Entity representation of features such as pockets are usually done by specifying the faces of pocket and a host of other parameters. We will define the pocket by its linear parameters and relative position to its constituent face.

3.9.1 Semantic Definitions and operations

The definition of pockets is related to the definition of slots. The difference between the slot and pocket operations are in three parameters - orientation, offset and number of blocks produced from one pocket operation (Figure 3.5).

As a generalized description, the following attributes describe a pocket adequately:

1. Face: Datatype contains the base block and face number following a standard scheme of numbering.

   face : block-id X Nat

2. Pocket-Depth: Specifies the uniform depth to be removed forming a pocket.

   pd : Real

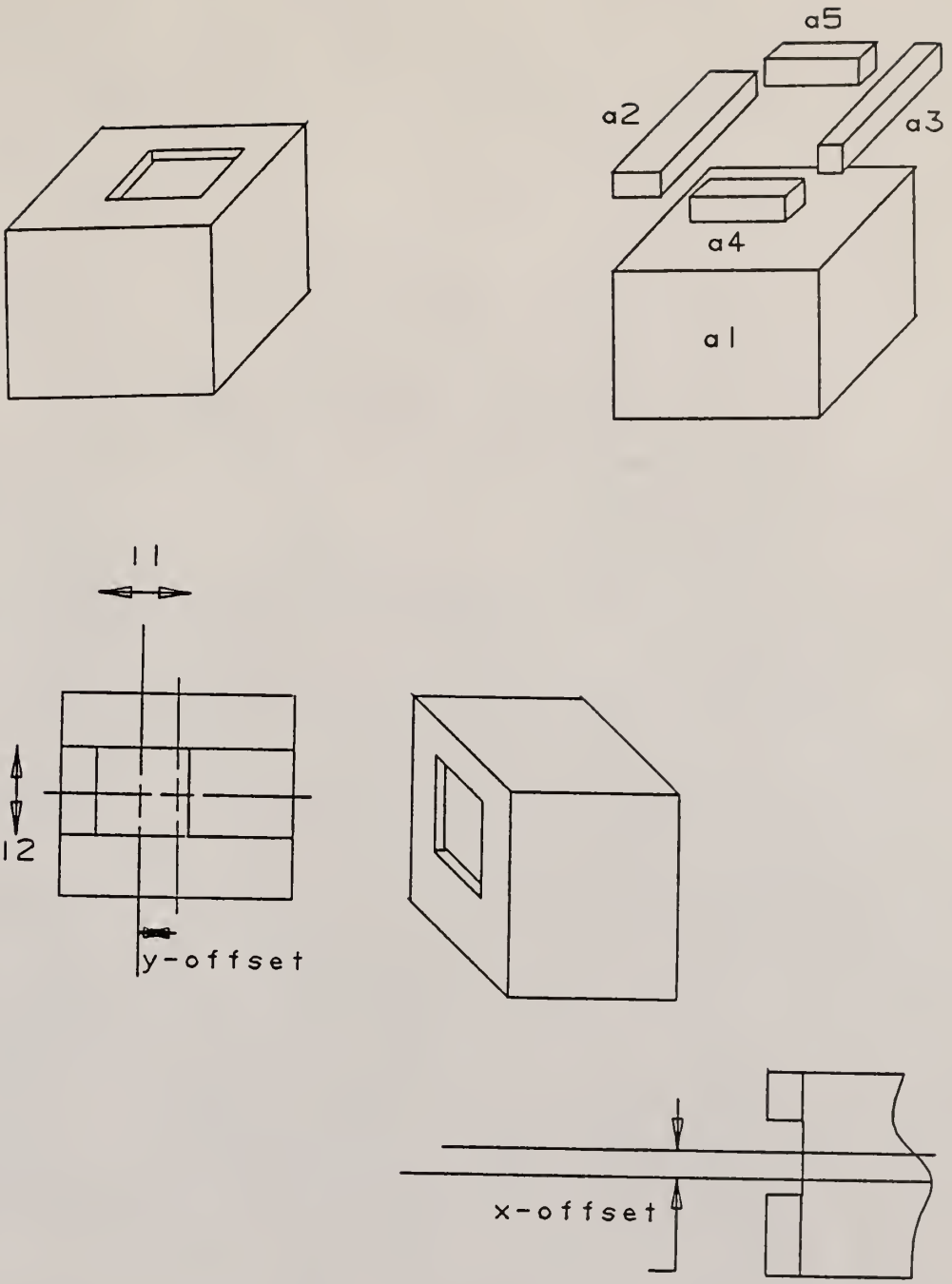


FIGURE 3.5 RECTANGULAR POCKET OPERATIONS

3. Pocket-Offset: Contains two attributes of offsets, the ordering scheme followed is:

faces 1 and 2:

offset1: y-axis offset

offset2: z-axis offset

faces 3 and 4:

offset1: x-offset

offset2: z-offset

faces 5 and 6:

offset1: x-offset

offset2: y-offset

This scheme allows any possible orthogonal orientation of the pocket and will represent any linear offset of the center of the pocket to the center of the face. Using these parameters, a generalized scheme has been described.

4. Pocket-Size: The pocket size can contain two of the three real number attributes: lx, ly and lz representing the dimensions of the pocket along any face. For example a pocket on face 1 (which is at right angles to the x-axis and lies in the y-z plane) is represented using ly and lz. These attributes, combined with the environment represents all possible pockets.

```
Semantic Algebra:
pocket-offset : po ϵ pockoff
      = Real X Real -> List
pocket-offset = α(offset1, offset2).
           (offset1, offset2)

pocket-size : ps ϵ pocksize
          = Real X Real X Real
pocket-size = α(x-len, y-len, z-len). (x-len, y-len,
                     z-len)

Pocket-in-Face : Face X Depth X pockoff X pocksize
Pocket-in-Face =  α(f, pd, po, ps).  cases of cdr(f)
         isNat(p)  -> {
     copy-block(car(f), car(f).old)  ->
       rename-block(car(f).[1], car(f)) ->
     slab-cut-block(f, pd)  ->  { p=1  ->
   create-block(e, car(f).[2], pd, ref(f,3),
    (ref(f,4)/2 + car(po) - ref(ps,3)/2),
     (ref(f,5) + ref(f,2)/2 + pd/2), (ref(f,6) -
    ref(f,4)/4 + car(po)/2 - ref(ps,3)/4),
         ref(f,7))   ->

   create-block(e, car(f).[3], pd, ref(f,3),
    (ref(f,4)/2 - car(po) - ref(ps,3)/2),
```

```
      ref(f,5) + ref(f,2)/2 + pd/2), (ref(f,6) -
    car(po)/2 - ref(ps,3)/4 - ref(f,4)/4),
    ref(f,7))4  ->

  create-block(e, car(f).[4], pd, (ref(f,3)/2 +
    cdr(po) - ref(ps,2)/2), ref(ps,3),
    (ref(f,3) + ref(f,2)/2 + pd/2),
      (ref(f,6) - car(po)), (ref(f,7) -
   ref(f,3)/4 - cdr(po)/2 - ref(ps,2)/4))
      ->

  create-block(e, car(f).[5], pd, (ref(f,3)/2 -
   cdr(po) - ref(ps,2)/2), ref(ps,3),
    (ref(f,3) + ref(f,2)/2 + pd/2),
    (ref(f,6) - car(po)), (ref(f,7) -
   ref(f,3)/4 - cdr(po)/2 - ref(ps,2)/4))
```

In the preceding semantic code for generating pockets in faces of blocks, there are four calls to the create-block function. The walkthrough of the pocket-milling operation and the transformation is as follows:

1. Select the face to be machined.

2. Describe the pocket to be machined on the face specifying the attributes described in the semantics.

3. At first, data processing operations are conducted to copy an instance of the block to be machined.

4. A slab cut is made with a depth equal to the depth of the pockets; this forms the first block of the compound that will represent the part after the operation.

5. The additional blocks are added automatically after extracting the length, width ..., origin information as a result of the operation performed.

6. The usual cleanup operations to remove non-integral geometry generated is carried out.

The result of these operations is the representation of the machining operation as CAD data by means of blocks. Similar reversal operations for undoing operations can be developed and implemented. The operations specified for machining are stored in a manufacturing list. When a call is made to extract information from this list, the environment database can be again transformed to represent parts after the undo operations.

3.10 EXAMPLE USES OF MANUFACTURING FUNCTIONS

The following example will illustrate how the manufacturing functions developed earlier in this chapter can be implemented. The objects COMP, C1, B1, B2, and B3 developed in the design example (see Section 2.10) will be used as the design geometry. The functions to specify material thickness around the parts will be illustrated. Machining the different faces of COMP using slab-cut and slot-cut will be illustrated with variations in geometry and recovery suggestions. Pocket milling is demonstrated using the simple block B1 with the variations in pockets.

1. Casting thickness around COMP

To specify a casting derived from COMP and with material added to the top face (face 6 of B1). Material will be added to face 6 uniformly after the calculations of inter sections. A new block with the casting thickness is added to the compound COMP resulting in the creation of COMP.cast.

cast-part(e, COMP, 0.2, f(B1, 6))

f(B1,6) specifies face 6 of B1. As a result of the above operation, COMP.cast is formed with new parameters found in

the comp-list in the environment(e). Additional cast-part operations are done to specify thickness on different sides.

```
cast-part(e, COMP.cast, 0.1, f(B2,2))
cast-part(e, COMP.cast, 0.3, f(B3,1))
```

These operations add casting material on the specified faces giving casting allowances of different thicknesses on the different faces of the blocks B1, B2 and B3 (see Figure 3.7).

2. Blank Specifications around COMP.

The following operation creates COMP.blank from COMP after specifying the blank specifications around the part. The maximum x, y and z parameters are found, COMP.blank will be added to the block list with parameters extracted from the object specified after adding the blank thickness.

```
blank-part(e, COMP, 0.5)
```

Note that the constituting blocks in COMP or copies of them do not transform, instead a new block, COMP.blank is formed (see Figure 3.8).

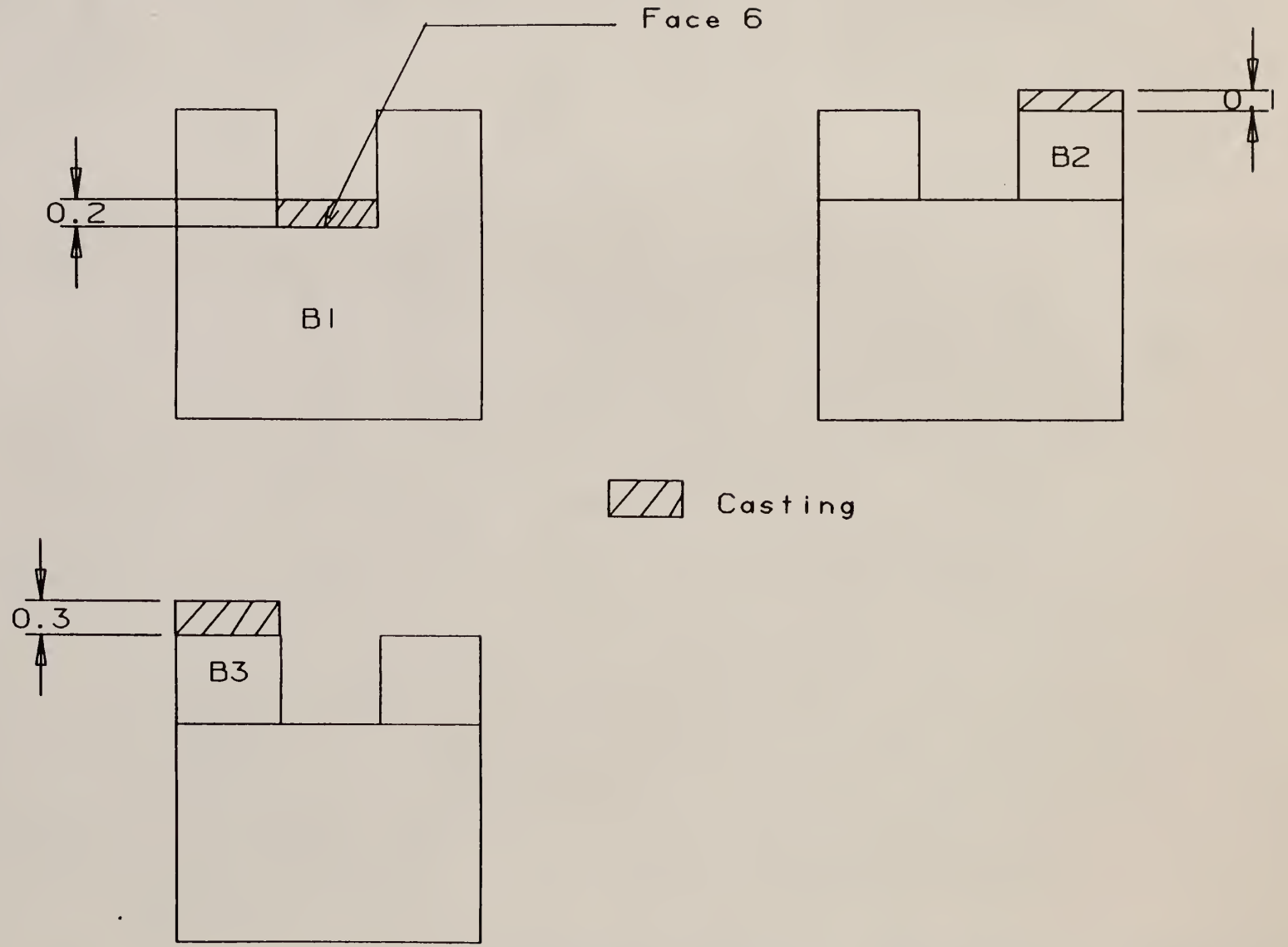


FIGURE 3.7 EXAMPLE 1 - SPECIFYING CASTING THICKNESS

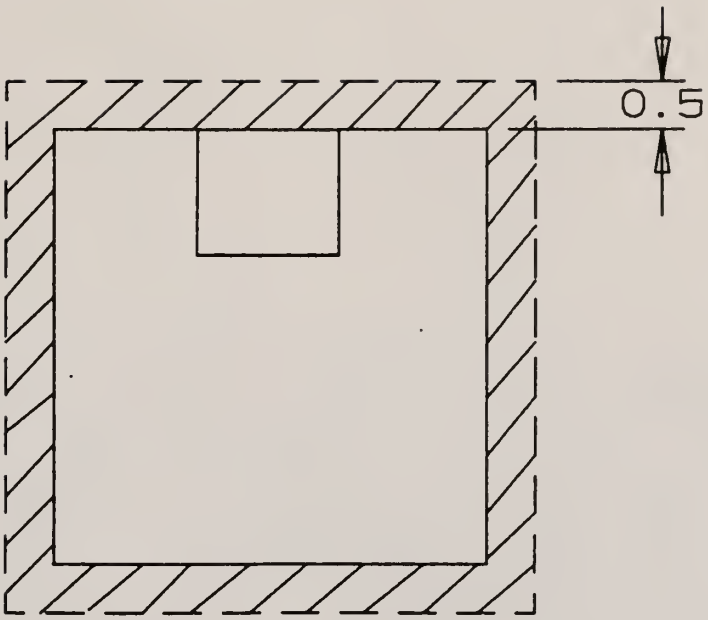


FIGURE 3.8 EXAMPLE 2 - SPECIFYING BLANK

3. Slab cutting examples on COMP and B2, B3.
(see Figure 3.9)

A slab-cut specified on a face of a compound or block is of the nature:

slab-cut(e, f(B2.cast, 6), 0.25, surf, t(f1,f2,0.02,0.01))

The above slab-cut performs a cut to a depth of 0.25 units on the face specified by f(B2.cast, 6) with the tolerances specified. The meaning to the tolerance t is the distance limited by the faces f1 and f2 has an upper tolerance of 0.02 units and a lower tolerance of 0.01 units.

f1: f(B2,6) and f2: f(B1,5)

Geometric transformations occur on the block B2.cast as a result of the slab-cut performed on face 6 of B2.cast, reducing the geometric attributes (in this case the height) and the origin of B2 to maintain symmetry of the body. Tolerance is added to the tolr-list as a result and the surface finish of the face is recorded. B2.cast is an integral part of COMP.cast and hence COMP.cast also changes as a result of the slab-cut on B2.cast.

A slab-cut operation performed on COMP.blank simply transforms the block parameters according to the face associated with the operation.

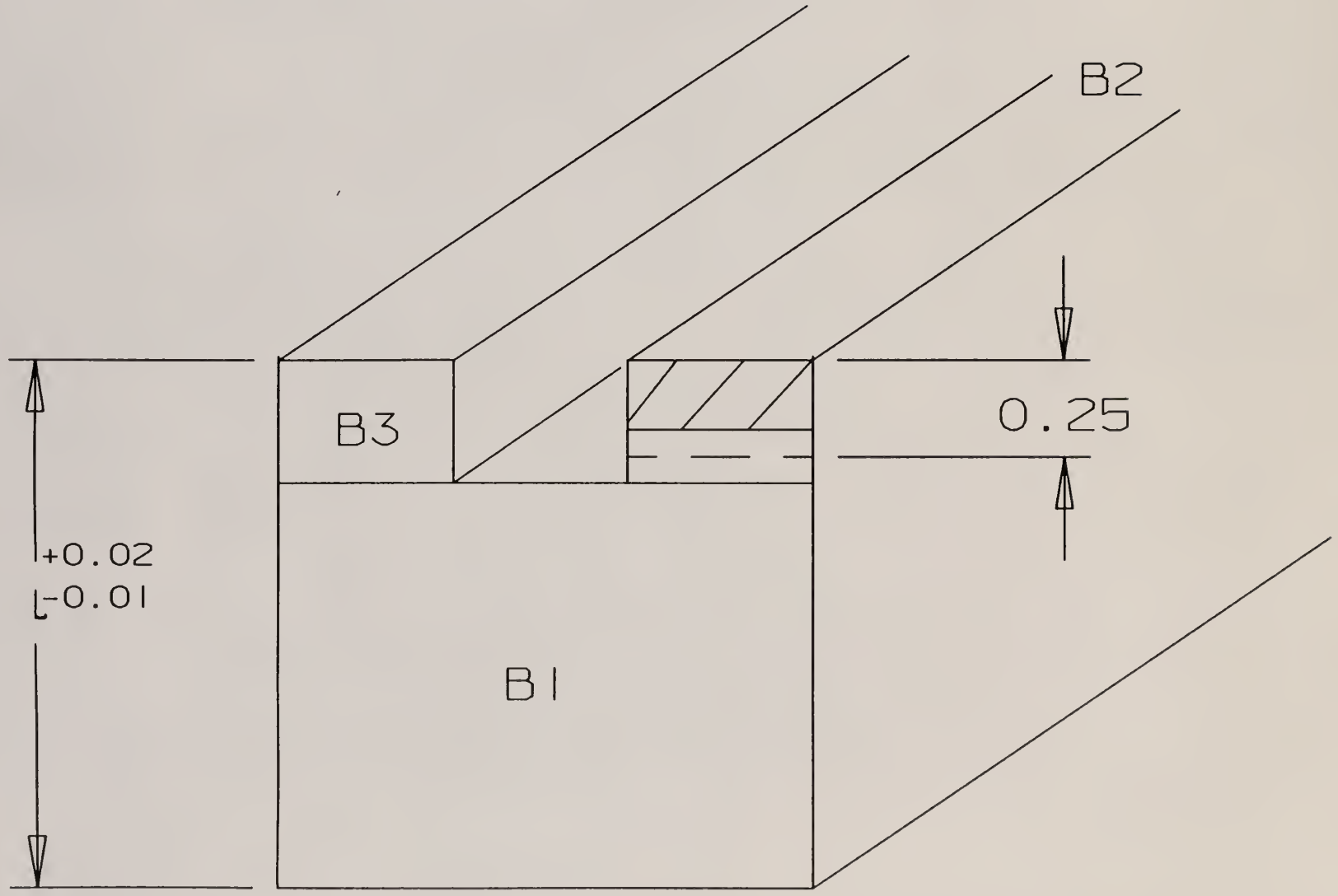


FIGURE 3.9 EXAMPLE 3 - SLAB CUTTING

slab-cut(e, f(COMP.blank, 6), 0.15, surf, t1)

4. Slot Cutting

Cutting a slot on the face of a block or a compound transforms the block parameters and reflects in the compound (Figure 3.10)

slot-cut(e, f(B1.cast,6), 1, 0.5, t1, t2, o(x-axis,.25))

Performs a slot-cut on face 6 of B1.cast, removing material from face 6 of B1.cast in a slot-cutting operation. The resulting geometry is composed of: B1.cast[1], B1.cast[2], B1.cast[3] after a slot-cut on B1.cast.

5. Rectangular pockets (Figure 3.11).

This example will demonstrate how rectangular pockets are formed on different faces and their transformations. As explained in the semantic equations, the effect of a pocket-in-face operation changes the block configurations. A compound is created with elements formed from the split ting of the base block due to the pocket operation.

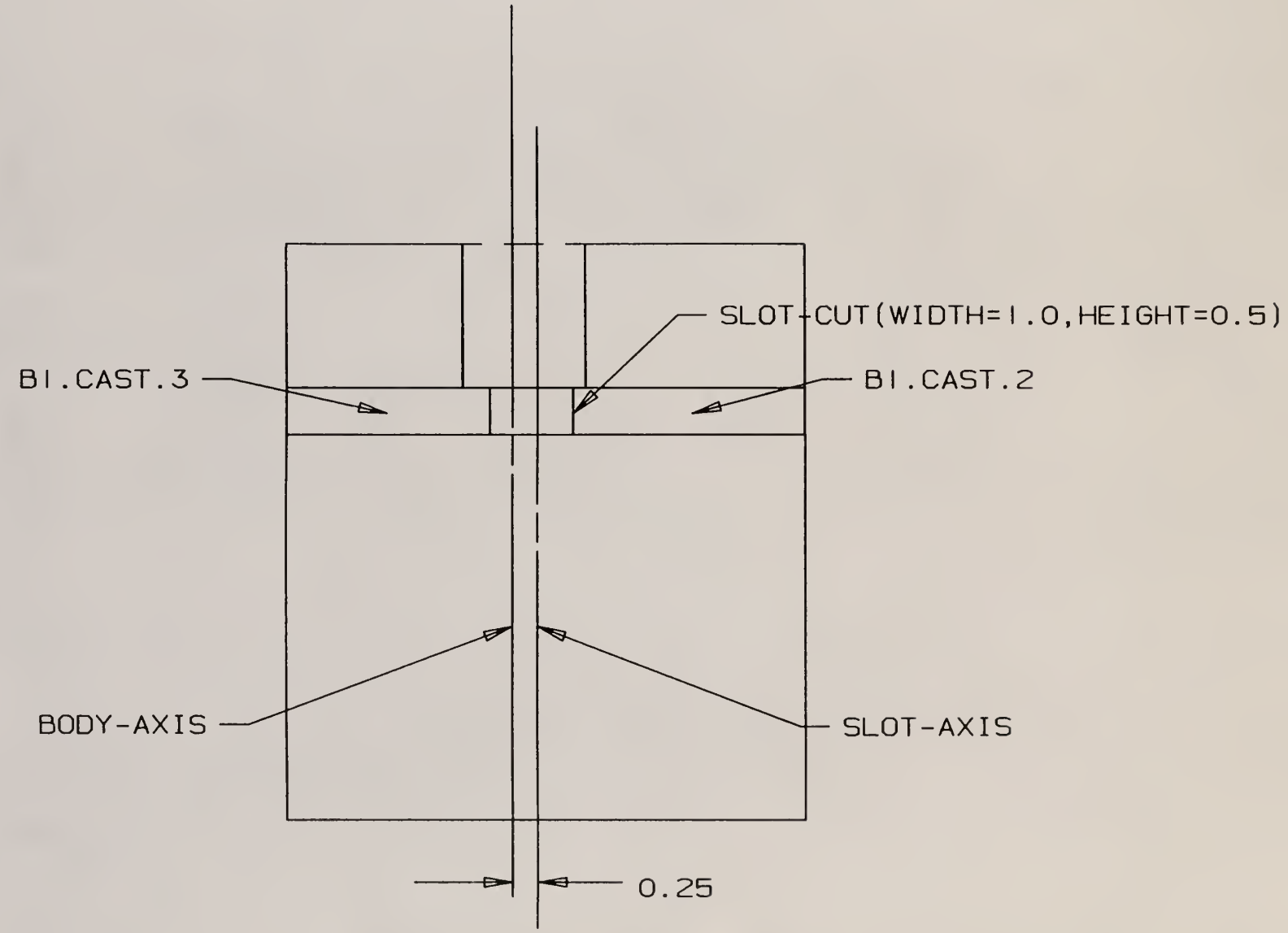


FIGURE 3.10 EXAMPLE 4 - SLOT CUTTING

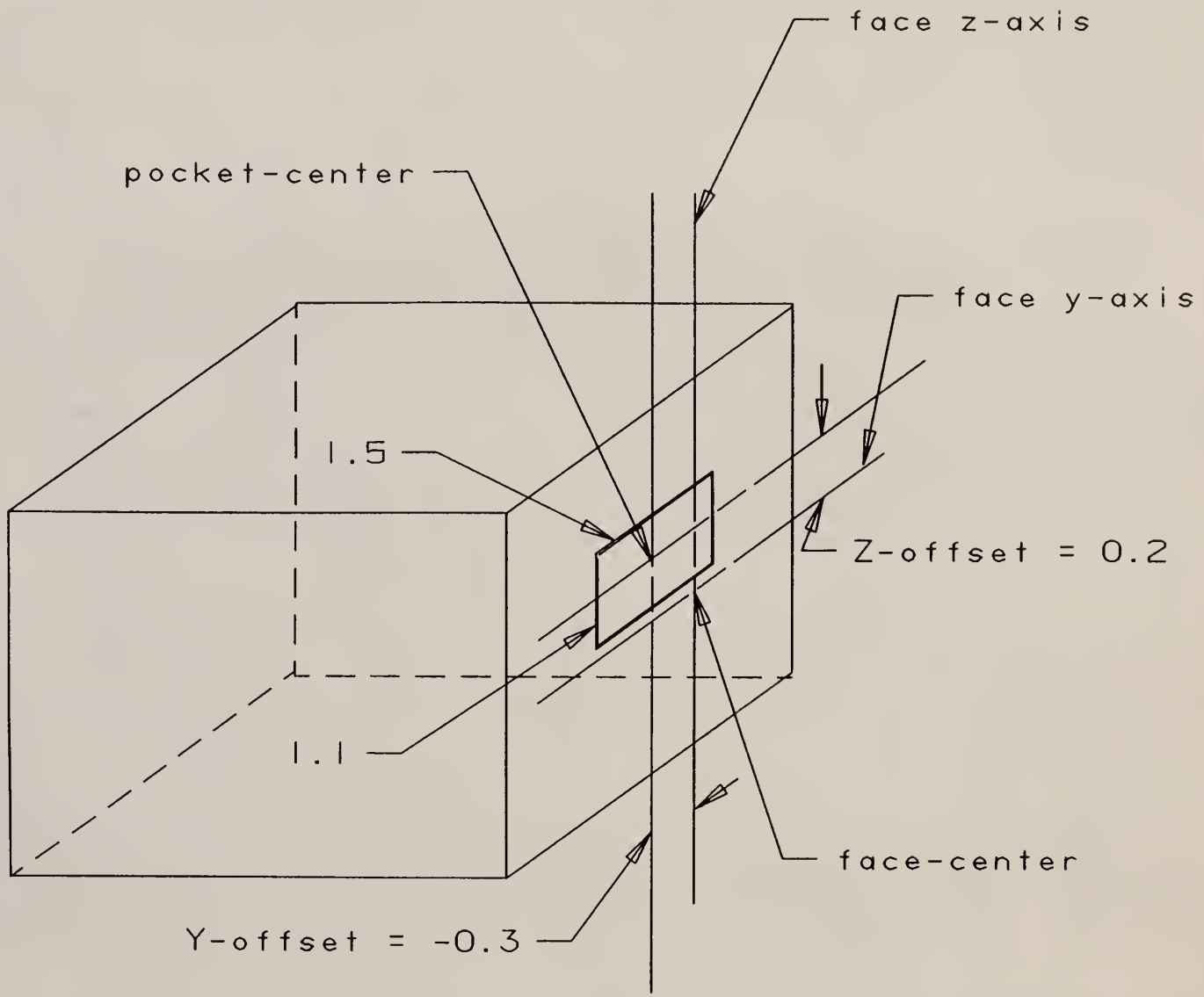


FIGURE 3.11 EXAMPLE 5 - POCKET MILLING

The offset parameters in the pocket-in-face operations are signed real numbers. The offset parameter is input in an ordered form, following the usual conventions of X-Y, Y-Z and Z-X schemes. Thus if a pocket is to be specified on face 1 or 4, the offset parameter 1 specifies Y-offset, offset parameter 2 specifies Z-offset. The same form is standard for all opposite faces.

Specifying a pocket on face 1 of block B1,

```
pocket-in-face(f(B1,1), 0.2, pockoff(-0.3,
                    0.2),ps(0,1.5,1.1))
```

The above expression specifies a pocket with pocket parameters as follows:

1. Y-length = 1.5
2. Z-length = 1.1
3. Y-offset = -0.3
4. Z-offset = 0.2

As a result of this operation blocks B, B[2], B[3] and B[4] are formed. They are grouped under the identifier B as a compound.

3.11 SUMMARIZING THE MANUFACTURING OPERATIONS

The major operations of defining a blank by means of a cast or extruded blank creates the geometry to be machined. The operations slab cut, slot cut and pocket mill present means to create complex parts in the geometric domain.

The semantics of manufacturing makes calls to the geometric semantics in order to create parts and for general data processing. Final geometry is represented by means of CAD semantics description. Similar to design, the operations developed in this chapter are low-level descriptions and have to be mapped to a user domain through interfaces.

4. IMPLEMENTATION CONSIDERATIONS AND EXAMPLES

Design and manufacturing operations of parts in a specific domain have been described thus far. This chapter will examine the capabilities of the system and discuss some key issues to make a working model of the conceptual representations and methods. Some implementation examples are also presented. Comparison of the actual system is made with the semantic system.

4.1 DATA STRUCTURING AND DYNAMIC MEMORY ALLOCATION

In this section we will compare two types of data structuring - binary trees and generalized lists from an implementation viewpoint. To represent members of a compound by means of lists, the programming language will have to support features such as variable sized argument declarations of objects and functions. Generalized lists seemed appropriate for CAD/CAM applications since the size of data elements is not known at the time of implementation or even at compile time.

Every manufacturing operation described gives rise to additional data objects. This demands dynamic memory allocation to be supported by the programming language. C++ and

LISP are two languages that support this feature naturally. C++ was chosen for demonstrating some examples due to it's object-orientation.

Binary trees require fewer functions than lists for general data processing. Search methods for binary-trees are trivial to implement. However, to construct a compound object from n blocks, the following relationships hold:

Binary-Tree:	requires 2*n - 1 operations before constructing a compound
Lists:	requires n - 1 operations to construct a compound.

Also, search methods using binary-trees consume more time than search methods for lists. Hence a List type of data structure is recommended rather than binary-trees for an actual implementation.

4.2 EXTENSIONS TO DOMAINS

The primitive domains described by the conceptual system are rectangular blocks. During the actual implementation, the domains can be expanded by programming language constructs. Object oriented languages (see Appendix), offer constructs known as class inheritance to assist in this. After a compound has been constructed (see Example 3.1), it can become a system primitive to derive others. Thus the level of domains increases by one. This new primitive set

has critical attributes for construction.

Example 3.1 - Class inheritance and the concept of objects

Let the primitive domain be a block - b(....attributes
initializers)

Construct an instance of the block b1 = b(...new attributes)

Construct b2 b2 = b(...new attributes)

Construct a compound c1 c1 = c(b1, b2)

Let us assume that b1 and b2 assemble to form a L-bracket c1 with thickness and length of limbs as attributes.

using a C++ keyword friend, c1(...new critical attributes
of c1) = friend b

After this stage, c1 can be used with the critical attributes as parameters to form a new L-bracket and so on.

4.3 USER INTERFACE

Most operations derived in the previous chapters are low-level to the system. A user interface to communicate the system capabilities and provide a friendly environment is a major requirement of implementation. An example user interface function will highlight this issue.

Example 3.2 - Ten attributes are required for the definition of a block. The first attribute, the block-id, is a character string representing the name of the block. The next

three attributes are the three linear dimensions. The next six attributes representing the body origin coordinates to the world coordinates can become confusing to input after several combine operations. Hence, the following steps will generate the origin attributes automatically:

1. User inputs a base block. The local origin defaults to the world origin if not specified by the user.
2. The user interface prompts the user to input the next block.

   There are two alternatives:

   - input second block separately.
   - input second block in relation to the position of the first.
     -- At this stage, several alternatives can exist such as face centered (the two mating block faces are centered), edge matched, offset by distance, etc. The list is long for specifying orientation of combinations of blocks.

Depending on the method of input selected, the origin coordinates and angles can be generated automatically. Such operations are key to building a friendly user interface.

In addition to menu input of user interfaces, graphical input user interfaces will provide an even friendlier environment. Icons, highlighting and graphic inputs (capturing

input information by moving an icon to the desired graphical entity and selecting) are the other tools for building highly efficient user interfaces.

4.4 MANUFACTURING FUNCTIONS

The manufacturing functions developed provide the means for defining blanks and basic manufacturing operations. Additional manufacturing functions can be defined and used by an applications programmer to generate parts. Perhaps the most important aspect of the presented manufacturing semantics is the ability to express manufacturing operations in terms of geometry. Evaluation of the design for manufacturability can be performed using a computer terminal to emulate manufacturing steps.

Application programs to extract data during machining operations can provide direct input to a CAM system. Additional manufacturing operations can be supplied as functions of cutting tools (selecting the cutting tool during time of specification from a tool-database) and cutting parameters. The attributes of such a manufacturing operation will be stored with the geometric parameters of component.

Numerical machine code can also be generated using the data described above. Also, a machine tool selection algorithm will generate process plans automatically. The machine tool selection algorithm will browse through the

cutting tools defined for a single component and determine the machine tool to be used. When parametrization of the availability of machine tools and other such data is done, the final system will be a complete process planner for manufacture.

4.5 EXAMPLE IMPLEMENTATIONS

Implementation of certain semantics were done using the C++ programming language in a VAX 11/750 hardware environment. Graphics were not generated. The modularity of the system and object-oriented methods assumed the availability of graphics codes in later developmental modules.

The basic data representation scheme and manipulation was the result of the functions developed. Language constructs in C++ allows the definition of classes of objects. Hence the basic datatype defined is object. The object class is overloaded (the same identifier can be used to denote objects with different attributes). At least two objects (blocks) should be created for an object (compound) to be created.

The attributes of the blocks and compounds are as described in Chapter 2. The Print function lists all characteristics of an object. Query function determines if an object is a block or a compound and calls the print function

to list object attributes. Delete and Modify functions follow the semantics of their corresponding semantic codes.

Object (compound) constructs a compound object combining the identifiers from block object 1 and block object 2. The data structure of the compound object is a binary-tree. The query function returns the number of nodes and all root elements (blocks) in the tree.

The Modify function modifies instances of blocks or compounds. The complete semantics described in previous chapters for manufacturing have not been implemented. Function calls to the modify block function will implement manufacturing operations.

The data_input module and other modules combine to make a user interface module. Alternate means of data-input for combining blocks/compounds to form objects are available. The origin for a new object is generated automatically.

4.6 A TYPICAL TERMINAL SESSION

This section will explain the operations that a user of the fully implemented system (incorporating the semantics and concepts) will follow, and trace the logic flow in the environment.

1. Start of terminal session, initialization of all environmental and system variables.

2. Opening menu:

   a. Design Functions

   b. Manufacturing Functions

   c. Applications Development

A few important functions have been listed; each function is modular and operates on the central environment database. Direct database manipulation functions can be described which write to disk, copy objects, etc.

3. User creates blocks and then combines them to obtain the designed part.

   Sub-Menus:

   i) Create Blank

   ii) Create Compound

   iii) Modify Blank/Compound

   iv) Query Block/Compound

   v) List All Objects Created

   vi) Add to Compound

   - Compound + Block

   - Compound + Compound

   vii) Delete From Compound

   - Compound - Block

   - Compound - compound

   viii) Display Functions

4. The design is passed to manufacturing engineering where the user creates a blank from which the part should be manufactured.

5. User starts machining operations to reach designed object. Specifies incremental machining cuts, surveys and queries machined blank as compared to design. Obtains geometric differences between design and blank in order to determine the size of machining cuts and operations to reach the final design. If a machining cut was specified erroneously the user changes the machining parameters to reach the goal.

6. Obtains match between design and manufacturing. Stores all machining operations made into file. This file will contain necessary tooling data for automated manufacture and stores part blank specifications for purchasing.

   Sub Menus:

   i) Create Blank
      - Casting
      - Rectangular Blank

   ii) Machining Operations
      - Slab-Cut
      - Slot-cut
      - Pocket-milling

iii) Specify tolerances

iv) Query status of blank with respect to design

4.7 SUMMARY

It has been attempted to present a picture of the proposed system as a practical and feasible CAD/CAM system. The above terminal session presented the major functions and capabilities of the integrated system as seen by a user.

During the actual part creation and operation history, three distinct phases are identified. These activities are performed by the designer and the manufacturing engineer. The three part phases would be:

1. Design Phase - part designer designs parts to final dimensions.

2. Blank Creation Phase - The manufacturing engineer retrieves the design created by the designer. A split screen session will display the designed part on one part of screen. The manufacturing engineer specifies raw material additions to the design. The raw blank appears on the second part of the screen.

3. Machining Process - The manufacturing engineer specifies machining operations. Changes are recorded and displayed on the raw blank. Intermediate queries will give the process status.

5. CONCLUSIONS AND RECOMMENDATIONS

The purpose of this project was to develop and formalize operations to automatically define manufacturing functions in terms of design variables. The key idea behind the development of these semantic equations was to explore the possibilities of designing a CAD/CAM system for limited geometry using abstract methodologies. The data and corresponding operations are extremely structured to represent abstract concepts using semantic notations. An additional feature of the semantic description system is that implementation is typically trivial after semantic descriptions.

Most computer languages have been developed on the basis of semantic descriptions. It seemed only right to describe semantically a new methodology attempting to solve complex situations in CAD/CAM applications. There were three major sections covered in this research. They were data description, design operations and manufacturing operations.

5.1 DATA DESCRIPTION

A formal methodology to describe the geometric attributes of rectangular parts has been developed and presented. The

basic object available to the user is a rectangular block. Using this basic object, a user can describe all geometry in the domain of orthohedral objects. The functions derived to describe data objects are versatile and carry periodic error checks. A data structuring methodology is also discussed. The structure adopted for this development was generalized lists. Object oriented data description is suggested to handle dynamic data in CAD/CAM applications. The basic data base management functions are described in detail.

5.2 DESIGN OPERATIONS

The primitive domain of this research project is orthohedral geometry. The semantic definitions to describe a basic object and derivations of this basic object have been described in detail. Functions to manipulate objects in a clustered form have been developed. They serve to modify objects, assemble and delete objects, and query object attributes. Additional CAD descriptions for transforming geometry have been derived. Most of the design operations have been developed for generic parts - the input can be simple blocks or compounds.

5.3 MANUFACTURING OPERATIONS

Manufacturing operations give the user important information regarding the method of manufacture. In chapters two

and three, the composition of any object in this environment is described. The action of any manufacturing function is to convert the overall shape to blocks. This is a general purpose methodology to describe geometry in the primitive domain described.

With additions and implementations to this project it is hoped to encompass a wider feature geometry to include solids bound by curves and general purpose prismatic geometry.

The functions to represent raw material allow a user to describe raw geometry in the form of castings or blanks which have machining allowances. The result of machining operations are converted to CAD geometry. Additional attributes of manufacturing such as tolerances, surface finish on the applicable surface, etc., are stored for post processing as input to CAM systems. These attributes are vital inputs to automatic process planning systems.

5.4 SHORTCOMINGS OF THE SYSTEM

The geometric domain is limited to orthohedral solids. In practice, geometry encountered will have many other features such as solids bounded by curves. In order to preserve the concept of designing and machining with solids, a new representation method for defining such geometry will

have to be sought. The semantics are fairly modular, additional features can be easily added without changing the philosophy of operations drastically. A user interface is necessary to implement the base system in a friendly environment.

5.5 FUTURE RESEARCH CONSIDERATIONS

The implementation of the concepts described in this paper is the next step. Broad guidelines have been presented for development. The semantic algebra gives an overall view of system design and will ensure consistent results regardless of the implementation language.

A short list of additions for future research is:

1. Expand the domain of primitive geometry to represent complex shapes.

2. Define more manufacturing functions in order to make machining operations accomplish the task of machining complex objects.

3. Adapt the system to automatic process planning by inferring design and manufacturing operations intelli gently.

Implementation considerations are presented in detail in Chapter 4. A modular approach towards development is recommended, object oriented programming techniques will implement reusable code. A short primer introducing object oriented programming is presented in Appendix.

A fully implemented system encompassing every form of geometry possible is the ultimate goal of any CAD/CAM system designer. This system will have as it's major features a kernel system describing all low-level data manipulation, a user interface system to communicate the capabilities of the kernel system and an applications programming environment to provide a user with tools to develop custom applications using methods already implemented. Such a system will automate the process of design and manufacture capturing and storing information vital to automated process planning.

REFERENCES

[1] Luby S.C., Dixon J.R., Simmons M.K, Creating and Using a Features Database, Computers in Mechanical Engineering, November 1986.

[2] Dixon J.R., Feature-Based-Design Research on Languages and Representations for Components and Assemblies, Design Theory '88, June 1988.

[3] Clancy J. J., Directions for Engineering Data Exchange for CAD and CAM, IEEE spectrum, May 1982.

[4] Su S.Y.W, Modeling Integrated Manufacturing Data with SAM * , IEEE journal, 1986.

[5] Beeby W. P., Integrating Engineering and Manufacturing, IEEE spectrum, May 1983.

[6] Smith B. M., et al, Initial Graphics Exchange Specifications (IGES) ver 2.0 NBSIR 82 National Bureau of Standards, 1982.

[7] Schmidt D. A, Denotaional Semantics - A Methodology for Language Development, Allyn and Bacon, 1986.

[8] Brooks S. L., Hummel K. E., Wolf M. L., X-CUT: A Rule Based Expert System for the Automated Process Planning of Machined Parts, United States Department of Energy, Bendix Kansas City Division, June 1987.

[9] Kramer T. P., Strayer W. T., Error Prevention and Detection in Data Preparation for a Numerically Controlled Machine, Proceedings of the 1986 International Machnie Tool Technical Conference, September 1986.

[10] Kramer T. P., Jun J., Software for an Automated Machining Workstation, Proceedings o fthe 1986 International Machine Tool Technical Conference, September 1986.

[11] Stroustrup B., The C++ Programming Language, Addison Wesley, Reprint 1987.

[12] Stefik M., Bobrow D. G., Object-Oriented Programming: Themes and Variations, The AI Magazine, December 1985.

[13] Koeing A., Static-Store Allocators for Variable Sized Objects, The C++ Programming Report, January 1989.

APPENDIX

THE OBJECT ORIENTED PROGRAMMING ENVIRONMENT

After the analysis of the several systems described in the previous chapters to represent data and operations in manufacturing applications, we proceeded to look for tools and means to implement our ideas. This appendix will discuss the object oriented programming environment as applied to manufacturing applications. The overall philosophy of feature-based design is very similar to that of object-oriented programming since both systems deal with clustered data that are related in some manner[12].

During the evaluation of different object oriented languages, C++ was found to be most suitable for applications in CAD/CAM situations. Apart from the ready availability of source C++ codes, properties of the language for efficient data manipulation, mathematical and scientific computation library functions make this programming language attractive for large scale applications. In addition, C++ code can be preprocessed to the C language code and can call standard library functions from other packages: C++ shares the same advantages as C in CAD/CAM applications.

Basic Concepts

Objects:

The basic concept in object oriented programming is the definition of an object: Entities that combine the properties of data and procedures. Objects consist of object class specification (object class names, instance variables, operators and parameters) and object class implementation (internal storage structures, procedures and functions that implement methods associated with data from the object class specification). Broadly, one can call objects user defined data types.

Uniform use of objects contrasts with the use of separate procedures and data in conventional programming. The separation of procedures and data gives rise to an important concept in object oriented programming: data abstraction.

Data Abstraction:

The class specification section (called private in C++) contains declarations of the data types and instance variables used in that class of objects. Object oriented programming languages do not allow users to access the data specified in the private section. This is commonly used to provide a high-level interface to the database management system. The most important point of data abstraction is

that the user need not know the implementation details to use the database and the associated database management systems.

Inheritance:

Instance values and operations of a particular class object can be related to an object belonging to another class. A sub-class can inherit the instance values and operations defined for its superclass. This can greatly simplify the database definition task and enhance the functionality of systems. Additionally, for very large applications, code can be reused, thus providing for overall performance and productivity. Multiple inheritance is another feature in OOP which is the property of inheritance of instance values and operations from many classes.

Multiple Definitions:

This is a unique feature of C++ to define procedures defining objects. Objects can be multiply defined (the keyword being overload). This means that procedures will define that a particular object will belong to more than one class. For example a convertible car can belong to a class of objects called automobiles and at the same time belong to an object class called vehicles. This feature is not common in many programming languages.

Design and Manufacturing Features as object classes

In engineering applications and CAD and CAM, objects can have distinct parts to hold compartmentalized information. An object holds design information as data fields and procedures for manipulating the data. A designed part will have the overall dimensions of the part (length, width and height for a simple prismatic block) and procedures to compute details such as the surface area and volume of part. Other procedures implement locations of edges and vertices.

Objects are divided into the two major subcategories of classes and instances. After an object (say, a baseplate with hole locations) has been defined as a class, instances of this class may be invoked, giving variations to data which attribute to the class. Such an instantiation is called constructing a class.

Thus for an object called baseplate, the private compartment will hold attributes of the base plate (geometric details and other component objects such as holes and their locations). The public part will have functions to construct an instance of the baseplate. The constructor initializes attributes of the private section of the class. Other functions will manipulate data in the private part such as query the object, error checks during construction,

storage and retrieval, to name a few.

Class Hierarchy

The hierarchical levels of classes are a desirable factor in object oriented programming. Compounded objects (objects combining to make another object) can be defined in OOPs in a hierarchical fashion. It is apparent that any assembly, subassembly or part going into an assembly can be defined as an object.

The programmer can design the system with basic primitives and give users necessary tools to create their own objects with ease. Using the baseplate example again, the user can first create a block (call it plate), create a hole (a subtractive feature) as two objects. He then combines these to form a compound class (baseplate) which holds attributes of block and hole. Going a step higher, the user defines other objects such as a L-bracket, jaw plate for a work holding fixture and combines these objects to form a final assembly. Methods of manufacture are procedures inherited from other class objects giving the final part rich data to describe design and manufacture.

Modularity and reduction of redundant data is very high in object oriented programming languages. The full attributes of any object being formed need not be supplied. Using the critical attributes for specification, the related

attributes are generated and stored.

A uniform interface to objects to provide a facility is called a pool of messages. Sets of related messages that perform a certain action is called a protocol. The manipulation of objects in a domain is a protocol and the manipulation of a particular object is called a message. Standardized forms of protocols allow objects of different classes to behave in the same manner[12].

Inheritance of class objects is a facility suited for CAD and CAM applications. For example if we have a baseplate as one class object and a L-bracket as another class object, a new object is created (say a T-bracket) having properties can be formed with attributes from both the base classes.

3.4 Operator Definitions to simplify programming

Operators can be defined in C++ to handle most CAD operations. Arithmetic operators such as the plus and minus sign are redefined to add two objects in a certain fashion. For example, to append an object to a parent in a node, the plus operator can used, specifying the parent, the left child and the right child. A new object is created. Similarly, a node is removed from a tree by specifying the node and the parent, and so on. In manufacturing, operations such as the addition of objects to a base (welding, joining, etc) can be

specified by the plus operator, giving additional attributes such as the nature of joining. Perhaps the most important operation in manufacturing, removal, is specified by the unary minus operator giving the specific identifier for the object face. Drilling (a hole) is specified as a user primitive object. This can alternately be specified by a minus operator.

Such operations when generically defined, form their own language for CIM applications. The user can step through the menu going through a set of procedures or use developed protocols and library functions as a front-end to build applications programs for specific environments.

A SEMANTIC MODEL FOR ORTHOHEDRAL PRISMATIC PART DESIGN AND MANUFACTURE

By

Srinivasan Sridhar

B.S. Mechanical Engineering
The University of Mysore, 1984

ABSTRACT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Industrial Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1989

ABSTRACT

The focus of this paper is to describe the development of a prototype system for the design of a set of prismatic components. A set of semantic functions to govern the set of orthohederal features has been defined and developed. The problems with part representation are presented in this paper. A semantic model of the system to design and manufacture components is developed and presented. The semantic model addresses key issues in the process of design and manufacture.

The primary work in this paper is the development of a semantic model to develop communications between the CAD and CAM parts of the system. This model describes automation of of the CAD and CAM parts of the system using common data models and interpretation of design and manufacturing operations.